SMALLTALK-72

INSTRUCTION MANUAL

Adele Goldberg and Alan Kay, editors

and

The Learning Research Group
Xerox Palo Alto Research Center

March, 1976

SMALLTALK-72 INSTRUCTION MANUAL

Adele Goldberg and Alan Kay, editors

SSL 76-6 March, 1976

The Smalltalk-72 instruction manual is intended for use by those persons with on-line access to the Xerox Interim Dynabook. The first two chapters consist of an introduction to some of the methods used for interacting with the Smalltalk system and for creating, editing, saving and retrieving Smalltalk programs. Chapter III goes deeper into the basic concepts from which everything else in Smalltalk is built. These include the method of evaluation of messages, message sending and receiving, and the notion of classes and instances.

Many classes have already been built for the user's convenience. These include the various classes for names, arithmetic, information storage methods, text display, and graphic control. The definitions of all of these basic classes is given in Chapter IV; Chapter V then presents a number of interesting examples that use these basic classes. Chapter IV also describes utilities already provided the user for editing definitions, saving and retrieving files of information, viewing definitions, testing values, and reading input devices.

Preface

The Smalltalk system described here was designed in the summer of 1972 and first conversed haltingly with a user late that fall. It was released for more general use at the Xerox Palo Alto Research Center (PARC) in spring 1973 when the first "Interim Dynabook" (a name for the current working version of a small computer system) became available.

This manual is intended for use by those persons with on-line access to the Interim Dynabook. As such, it employs a tutorial style that directs immediate experimentation with a Smalltalk system; it also maintains a somewhat informal dialog about expected results of such experimentation. There are references to peripheral devices, such as a keyset, a mouse, a display screen, and a disk, that have meaning mainly in the context of the Interim Dynabook. Furthermore, the manual references disk files that are needed in order to follow the suggested sequence for experimentation and provides information on how to obtain these files. Such information is only useful to those persons having access to the Smalltalk system library.

The purpose of making public an instructional manual about a language implemented on a computer not generally available is to ease the distribution of instructional information to school-age students (no younger than high school age) who will, in fact, have access to the Smalltalk system and materials noted here. Because an attempt is made to describe graphic results of running example programs, readers without access to the Smalltalk on-line materials may still gain some information about Smalltalk by browsing through these pages. Furthermore, the manual may assist these readers in developing their own experimental Smalltalk environment.

Many people (both from the *Learning Research Group* and from other groups at PARC) have worked hard to develop the systems described in this manual and accompanying documents--the design and implementation of the Smalltalk language, real-time music synthesis, animation, retrieval methods, color graphics, and network communications. We take space here to mention their names: Dan Ingalls, Chris Jeffers, Ted Kaehler, Diana Merry, Dave Robson, John Shoch, Dick Shoup, and Steve Weyer of LRG; David Boggs, Bill Bowman, Bob Flegal, Larry Tesler, Truett Thach, and Bill Winfield of System Science Laboratory; and Patrick Baudelaire, Larry Clark, Jim Cucinitti, Peter Deutsch, Ed McCreight, Bob Metcalfe, Mike Overton, Bob Sproull, and Chuck Thacker of the Computer Science Laboratory.

TABLE OF CONTENTS

PREPARING A BASIC SMALLTALK DISK

There exists a disk pack that contains the Basic Smalltalk System as described in this manual. To save on disk space, only the main files have been placed on this disk. These include the Smalltalk programming system including the windowing functions, an editing facility and printing routines, and some Smalltalk font files. Also included are files that contain the sample class definitions presented in the manual:

boxes, fontfns, nwindowfns, simpulafns, turtlefns,
windowfns, xydic, xfer, xyfns, xplot

Not included are all the files needed to run the music, animation, findit, and editfont frameworks. These can be retrieved onto your disk either (1) by transferring the files noted in the documentation on the various frameworks from a disk that already contains them, or (2) by executing one of the following (included) command files:

animationget.cm
finditget.cm
finditvget.cm
musicget.cm
editfontget.cm

The format for executing a command file is

@<filename>@ <return>

To update your files, either use a Basic Smalltalk disk for transferring files, or, if you have access to the archival file system, retrieve a file named

<smalltalk>smallmanual.cm

If you execute it as a command file, your disk will be updated with the Basic Smalltalk disk files listed above.

Chapter I.

INFORMAL ORIENTATION TO THE SMALLTALK SYSTEM

Introduction

This manual is intended for use by those persons with on-line access to the Xerox Interim Dynabook. As such, it employs a tutorial style that directs immediate experimentation with a Smalltalk system; it also maintains a somewhat informal dialog about expected results of such experimentation. Chapter I demonstrates some of the methods used for interacting with the Smalltalk system; it includes the use of display graphics, dialog windows, and font editing windows.

Chapter II continues this introduction by demonstrating methods for creating, editing, saving and retrieving Smalltalk programs. It then begins specific instruction on the development of Smalltalk class definitions, beginning with the class *box*, then expanding a box-shape into any regular polygon (the class *polygon*), and continuing with methods for communicating with instances of the class *turtle*. Included in this chapter is definition of the set of special symbols used in Smalltalk; some attention is paid to the idea of message sending and receiving. Finally, this chapter describes the class *dispframe*, and presents a number of ways to place text on the screen and to sketch with a "pen" and a "paint brush".

Chapter III goes deeper into the basic concepts from which everything else in Smalltalk is built. These include the method of evaluation of messages, message sending and receiving, and the notion of classes and instances. One part describes subsequent presentations of basic class definitions.

Many classes have already been built for the user's convenience. These include the various classes for names, arithmetic, information storage methods, text display, and graphic control. The definitions of all of these basic classes is given in Chapter IV; Chapter V then presents a number of interesting examples that use these basic classes. Chapter IV also describes utilities already provided the user for editing definitions, saving and retrieving files of information, viewing definitions, testing values, and reading input devices.

To Get Started

Place your Smalltalk disk in the machine, press "run" on the disk drive, and when the "ready" light appears (yellow light), press the "bootstrap" button (the little one located near where the wires enter the back of your keyboard). The screen will go blank for a second and then show you some information having to do with the particular machine configuration you are using. You are talking to the Interim Dynabook operating system. Type:

@s@ ⟨return⟩

@ is typed by holding down both the key marked 'SHIFT' and the '2' key. There will be a flash and a rectangle (window) will appear with text in it

A Smalltalk Window

If you are on a color machine (your screen background has color rather than white), you should type:

@cs@ ⟨return⟩

The Mouse

The little rectangular object with three buttons that usually sits to the right of the keyboard is called a mouse. Move it around while watching the screen. An arrow (mouse cursor) will be moving in response to it. This is how we point to objects on the screen. Smalltalk constantly "asks" the mouse where it is. A little bit further on we will explain how you can ask the mouse the same questions.

In Case of Disaster

In case of any disaster, first push the *⟨escape⟩* key (marked 'ESC' and located in the upper left corner of the keyboard). Try to put the mouse cursor in a displayed window or, by moving the mouse around, try to wakeup a "hiding" window. If that doesn't help, then try typing *⟨shift⟩⟨ctrl⟩⟨escape⟩*. That is, press the key marked 'ESC' while holding down the keys marked 'SHIFT' and 'CTRL'. Finally, as a last resort, press the "bootstrap" button again and go through the above sequence.

Talking To Smalltalk

If you are on one of our color machines then move the mouse so that the cursor travels all the way off the bottom of the screen. A white rectangle (a Smalltalk *dialog* window) will appear. It contains a message. Move the cursor into the window. If on a black-and-white machine, simply move the mouse so that the cursor travels into the rectangular frame at the bottom of the screen.

A small, flashing image of the Interim Dynabook will appear--this means Smalltalk is listening. To test this, type:

3+4 **!**

The **!** *⟨do it⟩* character is marked 'LF' on the upper right of your keyboard. It is used to tell Smalltalk that this is the message you really want it to do. Now try the following:

*3*4***!**

'*' is how we express the sign for multiplication in Smalltalk. Try:

355.0/113 **!**

The result shows a well-known number and the accuracy of Smalltalk's fractional arithmetic.

Helpful Notes

Smalltalk will only listen to you through a window when the cursor is in it. Any characters typed when you are out of a window will be saved until you place the cursor in a window. Try taking the cursor outside of the window and typing *3+4*. You will not see the characters appear in the dialog window. Now move the cursor into the window. The characters '3+4' will appear in the window. When you have learned to create multiple windows, you might repeat this experiment to prove to yourself that the characters will indeed appear only in the window containing the cursor.

Once you start typing characters in a window, Smalltalk will wait for you to type **!** before any window wakes up again. So, if you inadvertently move the cursor out of a window while you are typing, Smalltalk will continue to listen in that window.

Deleting (backspacing) of unwanted characters is done with the 'BS' key located on the upper right of your keyboard.

If you inadvertantly make an error of some kind, which is then sent to Smalltalk by saying ! (<do it>), a *diagnosis* window will appear with a message that, at this point, will probably be obscure.

To see this, try typing a symbol for which Smalltalk does not yet have a meaning, such as:

mumble !

A *diagnosis* window will appear. Note that the prompt character (the Interim Dynabook image) does not flash. Once a diagnosis window appears, it listens to you until you return to your previous context. To get back to your previous context, either type:

done !

or the shorter form:

<ctrl> D

typed by striking the 'D' key while holding down the key marked 'CTRL'.

Try A Turtle

Turtles are little beasts which crawl around on the screen and can leave a variable width tracing of where they have been. Smalltalk line drawings are done with turtles.

Smalltalk can have many turtles. Each is created as an instance of a group or class we call *turtle.* One, ☺ (called "smiley"), has already been created for you. It is typed by holding down both the key marked 'SHIFT' and the '2' key (i.e., the @ sign which has a different printing representation in Smalltalk than it does in the Interim Dynabook operating system).

As with all Smalltalk objects, ☺ can receive a variety of messages asking it to do "turtlelike" things (such as "go forward some number of steps", "turn some number of degrees", ...), and answer reasonable questions (such as "what kind of thing are you?", "where are you"). Type:

☺ *go 100*!	A vertical line should appear.
☺ *is ?*!	? is typed holding down both the 'SHIFT' and '6' keys. Is the answer (turtle) reasonable?
☺ *turn 90 go 100* !	Did what happened make sense?

To redo a previous statement, type:

redo n !

where *n* is the number of transactions (visible images of the Interim Dynabook) back from where you are. If you type: *redo 1*! at this point, the ☺ *turn 90 go 100*! message should be re-sent to Smalltalk and another line will be drawn on the display screen. If you want to redo the previous statement, simply type the equivalent statement:

redo!

Try

erase home!	Clears the screen, brings the turtle to its center position, and points the turtle upward
do 4 (go 100 turn 90) !	Will make a square
erase home. *for i ← 1 to 200 do (go i*2 turn 89)*!	To get a "squiral".

The text line change in the above transaction is obtained by pushing the key marked 'RETURN' after the message *home*. This "carriage return" does not affect anything except the appearance of the text in the text window. The period is a delimiter, signifying the end of a message. It is generally good practice to include periods when stringing together several complete messages. Note that, although the period signifies the end of the message, you still need to type ! *⟨do it⟩* to actually send the message to Smalltalk.

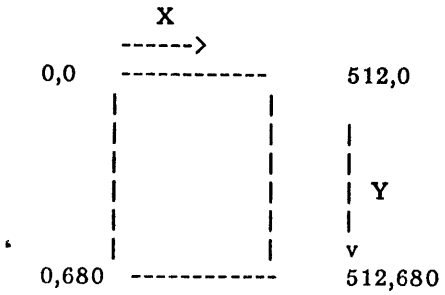
Notice that, as a result of the above messages, the black frame around the window has disappeared. The window has not been destroyed. Merely, 's drawing area overlapped with the window area, and hence erased much of the window information. None of that information is lost. Move the cursor off and then back into the remembered window area, refreshing the window display. This erases any part of the turtle drawing that overlaps the window. Any turtle lines inside the window will scroll (move up) whenever the text scrolls.

You have also just used two Smalltalk iteration methods: *do* and *for*. Each is a method for counting the number of times a message should be evaluated. In the more general method *for*, the iteration counter (in the above example, the counter is *i*) can be used as part of the message (in the example, *i* is used to help determine the distance the turtle will travel).

Layout of the Smalltalk Screen

The *x* direction runs from left to right. The left hand margin is *0*, the right hand one is *512*. The *y* direction runs from top to bottom. The top margin is *0*, the bottom one (at the lower boundary of the original window) is *680*.

Smalltalk display screen



Now say to the turtle:

erase! *goto 100 100*!	Is a line drawn to the top left quadrant?

Type *mx*. Then, before typing the !, place the cursor somewhere in the screen and type:

!

Similarly, try

my!

Smalltalk should send you back reasonable numbers for m(ouse)x and m(ouse)y, the display coordinates of the mouse cursor. Now type:

☺ *goto mx my*!

and a line should be drawn to the cursor position. You have hooked up the mouse to the turtle. A simple drawing program can be written by saying:

repeat (☺ goto mx my)!

Move the mouse and a trail will be left behind. You are in an "infinite" loop (the ☺ *goto mx my* will repeat forever). To escape from the loop and to get Smalltalk to listen to you again, press the key marked 'ESC' in the upper left hand corner of your keyboard and move the cursor back into the window.

Try

☺'s *width* ← *3. repeat (☺ goto mx my)*!

The 's is typed by striking the key marked 'S' while holding down the key marked <CTRL>.

A more involved drawing program might use the buttons on the mouse to control the turtle's ink color, width, and erasure. More about drawing programs later.

Dialog Windows

All communication to a Smalltalk object is done through *windows* which contain the most useful editor for that object (you have just been using a *dialog* window). The editor for a *picture* object is a kind of painting and drawing aid; the editor for a *paragraph* of text handles characters; the *font* editor allows the character defining dots to be easily changed; and so forth.

Every window can be moved, stretched, and deleted from the screen. Other abilities depend on the particular kind of window with which you are dealing. A collection of related windows (containing pictures, text) is a *document* which can be automatically *archived* in many different ways for later retrieval and editing.

For example:

a. Move. Move the cursor into the upper left hand corner of the window you are in and press down the top button. The window should go blank. You may have to play a little while holding down the button in order to find the actual corner. The tip of the cursor (the upper left corner) must be in the window corner.

b. Now point the cursor somewhere else on the screen and push the top button briefly again. The window will reappear in the new position. The upper left corner of the window can not be forced off the physical display screen; however, the other parts of the window can be slid off the display as a method for pushing them out of the way until needed again.

c. **Grow.** Now move into the lower right corner in a similar manner. (If the corner is off the screen due to the previous move, do another move further to the left to get the right hand side visible again.) The next button push will change the boundaries of the window so that the new lower right corner position will coincide with the cursor. Try it. You can not grow the window smaller than 32 units wide or 32 units high.

d. **Create.** A new *dialog* window will be created for you by grabbing the lower left corner of an existing *dialog* window (pointing the cursor and pressing the top mouse button). The new window will appear in the upper left portion of the display screen.

e. Position the cursor inside the new window and try typing *3+4!* .

f. **Delete.** Any *dialog* window can be deleted by grabbing its top right corner. Try it with the new *dialog* window. For obvious reasons, a single remaining *dialog* window can not be deleted.

Try overlapping windows. The window that sees the mouse cursor wakes up and displays itself on top of all other windows.

Each new *dialog* window appears in the upper left portion of the display screen. Unless you move each window as it is created, the windows will pile on top of one another. Another way to define a *dialog* window is to have a new window appear at a location pointed to by the mouse cursor. The cursor could blink on and off, waiting for you to press a mouse button to indicate that the present cursor location is the place to put the new *dialog* window. Later on, after you have learned more about Smalltalk, you might make this change to your personal Smalltalk system.

A First Note on Smalltalk Classes

Every entity in Smalltalk's world is called an object. Objects can remember things and communicate with each other by sending and receiving messages. Each example we present demonstrates the ability of objects to receive messages and produce replies.

Every object belongs to a class (a method for grouping together objects that do similar things). 🐢, for example, is an object. It is a member of the class *turtle*. All members of this class are able to draw lines on the display screen. The class handles all communication (receiving messages and producing replies) for every object which belongs to it.

We have just been looking at members of the class *window*. Messages are sent to a *window* by pointing with the mouse cursor and pressing a mouse button. Each member of the class responds to the message by moving to a new screen location, changing its size, creating a new member of the class, or deleting (erasing itself from the screen). The objects are dialog windows, capable of capturing and editing Smalltalk messages. The next example is a *font* window which contains an editor for designing display characters.

Font Editing Windows

Type

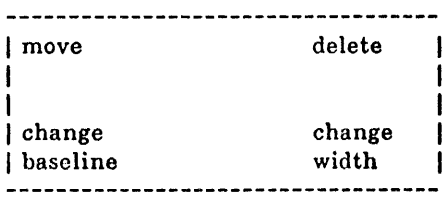
filin 'fontfns'!

filin is the Smalltalk method for reading messages stored on a disk file. Reading the file takes a while. The display screen is purposely turned off (becomes blank) to speed up the reading process.

You now have routines for creating windows in which editing means drawing in a matrix of black and white dots. These windows contain magnified views of display characters. Any character font (the design of the display characters) can be described as a matrix of black and white dots. Using the mouse cursor in a *font* window, you can draw in a character font of your own choosing. Moving the cursor to a dialog window, you can immediately view font changes within the context of text displayed in that dialog window. Type

fontchar!

A newly created window appears in the upper left corner of the display screen. Like dialog windows, a *font* window can be moved, deleted, and its size changed. Unlike dialog windows, a new *font* window is created only by typing the message *fontchar*.



Four actions are taken by pointing to one of the corners of a *font* window and pressing the top mouse button.

1. **Move the window.** Point to the upper left corner and press the top mouse button. Then point to a new position on the display screen and press the top mouse button.

2. **Delete the window.** Point to the upper right corner and press the top mouse button.

3. **Change the baseline of the character.** Point to the lower left corner and then to the relative adjustment, up or down, of the character's baseline. Raising the baseline creates superscripts; lowering the baseline creates subscripts. The upper limit is the baseline of the previous text display line; no lower limit exists with the exception that an attempt to print outside the display screen boundaries will cause Smalltalk to crash. Note that the *font* window appearance does not change; the change only appears in the printed text. Move the cursor into the dialog window to see the change.

4. **Change the width of the window** (and, thereby, the width of the matrix). Point to the lower right corner and then to the new right margin. The width is rounded to a multiple of 16 display bits and may not exceed 16 dots, so it may not appear exactly at the mouse cursor's arrow head.

5. **Drawing black and white dots.** Black dots are painted into the matrix by pointing to a location in the window and pressing the bottom mouse button. The drawing technique is to scratch black lines through the matrix dots as long as the mouse button is pressed. As soon as the button is released, the black dots appear in any area containing the black lines. White dots are painted by pointing to a location in the window and pressing the middle mouse button. White lines are written through the dots as long as the mouse button is pressed; white dots appear when the button is released.

6. New characters. When the window is first created, the character available for editing is the period, '.'. To change the character, place the mouse cursor inside the window and type, on the keyboard, the desired character.

Once a new font has been designed, it is saved on a disk file by typing

filfont ⟨filename⟩ out !

where ⟨filename⟩ is some name delimited by single quote marks. For example,

filfont 'myfont' out !

The font of the dialog window in which you are currently typing is the one that will be saved.

To read a saved font, type

filfont ⟨filename⟩ in !

For example,

filfont 'myfont' in !

The font of the dialog window you received when you first started working is stored on a filed named

st8.al

If you have made changes but would like to return to the original (default) Smalltalk font, type

filfont 'st8.al' in!

Other Smalltalk fonts include st6.al and st10.al; each can be retrieved from the archival file system.

The font of the dialog window in which you are currently typing will change to the font saved on *⟨filename⟩*. The font you edit is the one currently belonging to the dialog window in which you are typing. Note however, that each dialog window is created with references to the identical font. In order to have two *font* windows editing separate fonts for each of two dialog windows, it is necessary to replace one of the dialog window's font with a copy of itself. For example, suppose there are two dialog windows (A and B) and suppose you type *fontchar*! in window A. Results of editing the single *font* window will appear in both A and B. Now type in window A

fontchar font disp's *font*!

Recall that the 's is typed by stiking the key marked 'S' while holding down the key marked ⟨CTRL⟩.

The class *fontchar*, upon receiving the message *font*, will replace the font for dialog window A with a copy of the value following the message *font* (in this case, with a copy of the font possessed by A). Results of editing the new *font* window will then show in A and not in B; moreover, results of editing the original *font* window will show only in B. Choice of which fonts are saved will depend solely on which window is used for typing the *filfont* message.

The use of the name *disp* and the message 's are described in more detail in subsequent sections. For now, assume their use for the above redefinition of a dialog window font.

Warning: some fonts have no definition for the character whose Ascii code is 31. This is the character used to mark the black dots. Any font without this character properly defined can not be used with this font editing system.

Chapter II. WRITING SMALLTALK PROGRAMS

Simple Manipulation of a Simple Program

To hand an object 'd' the meaning '3' in Smalltalk, we say:

☞d ← 3!

(The ☞ is typed as <shift> '). If you now say:

d!

The meaning (or value) of *d* (which is a number, *3*) will be returned.

Each object in Smalltalk can only have one meaning. To change the meaning of the object named 'd', we might say

☞d ← turtle!

The new meaning (or value) of *d* (which is a turtle) will be returned.

In these examples, we use the symbol ☞ to indicate that a literal name follows. The arrow, ←, indicates a desire to give the name a meaning.

Previous *turtle* examples showed how we can get a *turtle* to draw a square. Now we need to be able to make that definition a Smalltalk object, use it, change it, save it, and retrieve it. To do this we need to give a name to the **actions** which cause a square to be drawn. In Smalltalk, **actions are also objects**. So we need to say something similar to what was just said to *d*. Type:

to square
(do 4 (☺ go 100 turn 90)) !

This will cause Smalltalk to give the actions *do 4(☺ go 100 turn 90)* the name *square*. Here, the symbol *to* (rather than the hand ☞) indicates the desire to give a name to some actions; the actions are enclosed in parentheses.

Erase the screen and bring the *turtle* back to home position by saying:

☺ erase home !

Then say:

square !

The stored actions will be invoked. The commonly used actions of clearing the screen and telling the *turtle* to go to home can also be abbreviated:

to cl (☺ erase home) !

Now only 3 characters have to be typed:

cl!

rather than 13.

Now type:

defs!

A list of the names *square* and *cl* should be typed back at you. *defs* is a kind of "bushel basket" which contains the names of user-defined programs.

How to Edit Your Definition

In any dialog window, type:

edit square !

An editing window with a command menu will appear. The "method" of *square* is shown as:

do 4 ()

The () stands for a parenthesized message which in this case contains:

 go 100 turn 90

Actual parentheses never show in the editor, only the marker () indicating levels of parentheses. To see the message within the parentheses, point the cursor at the word 'Enter' in the menu and push the top button on the mouse. (Note, some versions of the mouse have buttons laid out horizontally, left to right, rather than vertically, top to bottom. Henceforth, we will refer only to top, middle, and bottom buttons; the left button corresponds to the top button.)

You should see the message as:

 go 100 turn 90

Place the cursor on the word 'Leave' in the menu and press the top mouse button. You have now backed up to the next higher level of parentheses.

We will use the word "grab" to stand for the compound operation of positioning the cursor on an object (word, icon) and pushing a button on the mouse to tell the system that the object we are pointing at is really the one we mean. (Unless specifically stated to the contrary, push the top mouse button).

Grab 'Enter' again.

Now let's change the *100* to a *50* in the definition of *square*. Grab 'Replace'. It will reverse its display color to show that the selection is understood.

Grab '100'. The top half will reverse color. This means that 'Replace' expects you to replace one or more elements beginning at '100'. We only want one element, so grab '100' again. The bottom half will also reverse color and a prompting Interim-Dynabook image will appear, indicating that typing is expected. Type:

50!

You will now see:

 go 50 turn 90

Now grab 'Exit' to terminate the editing context. You will be returned to the previous Smalltalk context. Say:

square!

and one of size '50' will be drawn. So the "meaning" (or "actions") of *square* has been changed.

A Note on Editing

There are a number of ways to terminate an editing sequence before completion. If you grab a wrong menu word, or have not completed the selection of a phrase to replace or delete, you can terminate by pointing the cursor outside the editing window and pushing the top mouse button. This does not work for 'Add', 'Insert', nor 'Exit'. If you do not want to complete an add or insert command, but have already received the Interim Dynabook prompt character, just type ! (i.e., insert or add nothing). Once you have selected the phrase, a replace command cannot be terminated unless you are willing to lose any previous edits. Pressing the 'ESC' key takes you out of the edit window and back to the dialog window. Also note that if there is more than one parentheses marker displayed in the edit window, the 'Leave' and 'Enter' commands expect you to point at the appropriate marker.

Generalizing the Definition of Square

Now suppose we would like to make *square* more general, so that it will draw squares of any size. To do so we can give *square* a "message" saying what the size should be this time, such as:

square 150 !

We must now change the definition of *square* so that it can receive the message and act accordingly. First say:

show square !

to remind yourself what the current definition of *square* is. We see:

```
to square
  (do 4
    (☺ go 50 turn 90))
```

It's clear that we want to do something with the place where *50* is. Everything else about the definition (having 4 sides and turning 90 degrees) describe squares in general.

Suppose there is a way to receive a value from the message. The value needs to be some number. We give the particular value a "name" in order to talk about it since we don't know beforehand what the number will be. Let's call it *size*. Looking above, we see that *size* should replace the *50* :

```
to square
  (do 4
    (☺ go size turn 90))
```

Now we just need to get *square* to receive the value of a message and call it *size*. In Smalltalk, the request to "receive the value of a message" is expressed by a colon

:

So we want to add

size ← :.

to the beginning of *square*. Say:

edit square!

Grab 'Insert', grab 'do', type:

size ← :.!

Careful--the period is necessary here. It helps to separate, in one's mind, the sequence of receiving a message and then invoking an action for producing a response. Note that the 'Insert' command inserts before the selected element.

To replace the *50*, grab 'Enter'. You should see

go 50 turn 90

Grab 'Replace'. You want to replace the '50' so grab '50' and grab '50' again (indicating the beginning and ending of a phrase to be replaced by new text). Now type the new text

size!

Grab 'Exit'. You are no longer talking to the editor. Type:

show square!

to see what you've done. It should look like:

```
to square
  (size ← :.
    do 4
      (go size turn 90))
```

Then try sending several messages to draw different squares:

square 150!
square 10!

and so on.

The colon expresses a request to Smalltalk to fetch the next value in the message. The value is the meaning of the next object (for example, the number 10). But the value can also be the result of actions taken by the next object. For example, try

square 150+20!

Smalltalk runs the definition of *square*. When it sees the colon in (*size ← :.*), Smalltalk "activates" the next object, the *number 150*. This number sees the plus sign (+), fetches the value of the next object (in this case, the *number 20*), and performs the addition. The value returned as the value of *size* is the sum 170.

The definition of *square* is obviously working but is a bit untidy. To see why, type:

size!

The value of the last size you gave *square* will be returned. This shows that the "name" of the message for the *size* of *square* belongs to everyone. It is much better for *size* to belong only to the object which uses it. To do this we only need to tell *square* that *size* belongs to it by putting the name *size* right after the name *square* in the "title" part of the definition. Say:

edit square title!

square's title line will be shown as well as (), the marker representing the body of the definition. If you were to 'Enter' (), you would see the definition itself. Instead, grab 'Insert', grab (), type:

size!

Grab 'Exit'. Type:

show square!

You should see:

```
to square size
  (☞ size ← :.
   do 4
    (☺ go size turn 90))
```

Later, when more of the Smalltalk system has been explained, we will adopt some abbreviations to make our story more compact and clear. For example, a short way to talk about this program would be to exhibit, in a general way, what has to be said to get results:

square ‹number›!

means the object *square* expects anything which evaluates to a number as a message. An example might be

*square 30.4+(111.7*65.789)/99*!

Here, the colon in (☞*size* ← :.) fetches the result of the expression 30.4+(111.7*65.789)/99. This example demonstrates the left-to-right method for receiving messages; that is, Smalltalk first sees the floating point number 30.4 which, in turn, sees the plus sign and attempts to receive a floating point number for the augend. However, the arithmetic is right associative. The augend is obtained by fetching a value from the message. As a result, the floating point number (111.7*65.789) is evaluated which, in turn, sees the division sign and requests a divisor (the 99.). Hence, in this expression, the multiplication is carried out first (because of the explicit parentheses), the division second, and the addition last. Try

10 - 5 + 2! response is 3, not 7

or

*20 - 2 * 3*! response is 14, not 54

Fixing Your Dialog

You can edit the command lines (or statements) in the dialog window in the same manner that you edit a named definition (described in the previous section). To fix a previous command line, type:

fix n !

where *n* is the number of transactions (visible images of the Interim Dynabook) back from where you are.

An editing window with a command menu will appear. After making changes, you grab 'Exit' to terminate the editing context. This causes the edited line to be sent and evaluated as a message to Smalltalk. The line in the dialog window will not be altered.

Saving and Retrieving Programs

Type:

defs!

again. *square* and *cl* will still be there. To save everything in *defs*, type:

filout ‹some name in single quotes›!

such as:

filout 'mysquare'!

The screen will go blank for a second.

To test whether you actually saved them, go through the "To get started" sequence again. Then try:

square 100!

This will generate a *diagnosis* window with the complaint that "square has no value" . We are now in a "clean" version of Smalltalk, one in which *square* has not been defined.

Diagnosis Window

The complaint is stated in a *diagnosis* window. Smalltalk attempts to state the complaint and then (1) to provide the name of the program in which the complaint occurred, and (2) to point, with a big arrow ➳, to the object causing the problem.

In the context of the diagnosis window, you can type any Smalltalk messages. The value of objects are within the context of the object in which the complaint occurred. In the above example, we are still at the "top level" of Smalltalk; that is, the context is a global one for all objects defined in Smalltalk. Each attempt by one object to evaluate another object takes you one level lower in context; after completing the evaluation, you return to the object that requested the evaluation at its higher level of context. It is possible to trace back from the current context in order to locate the cause of complaint. Each time you type

c !

you see the next higher level of context.

Type

done! or *‹ctrl› D*

to get out of the diagnosis window.

Now type:

filin 'mysquare'!

After a few seconds, try:

square 100!

The result shows that you have retrieved your program.

Type

size !

You will get a complaint that "symbol has no value" because now *size* only belongs to the object *square* that uses it. The object *size* has no value in a more global context.

Special Characters

Smalltalk uses a number of special "iconic" characters, many of which were invented by some Smalltalk students to help remind them of important distinctions. An example is "quote" whose sign to adults is usually ("). The children preferred to use (☞) to signify a literal symbol, since in its typical use:

☞*joe*

(meaning the literal symbol 'joe' rather than what or who 'joe' may stand for)--the hand points directly at the symbol itself.

This distinction exists in English also. We can say:

Paris is a large city in France.

We shouldn't say:

Paris has five letters.

but rather:

'Paris' has five letters.

to indicate the literal word rather than the city.

Keyboard Equivalents

(Note, there are usually several ways to type a special keyboard character. The following table presents the methods most commonly used.)

To Get	You Type	We Call It
!	LF	do it
☞	<shift> '	hand
☌	<shift> 5	eyeball (look for)
⚷	<ctrl><shift>;	
⚷	<ctrl> k	keyhole, "peek"
⇒	<shift> /	if ... then
⇑	<shift> 1	return
☺	<shift> 2	smiley
☐	<shift> 7	
?	<ctrl> ?	
's	<ctrl> s	
done!	<ctrl> d	
-	<shift> -	unary minus
≤	<ctrl> <	less than or equal
≥	<ctrl> >	greater than or equal
≠	<ctrl> =	not equal
%	<ctrl> v	percent sign
@	<ctrl> 2	"at" sign
!	<ctrl> 1	explanation
"	<ctrl> o	double quote sign
$	<ctrl> 4	dollar sign

Summary of Special Dialog Window Operations

<esc>	Escape to the "top level" of Smalltalk; should return youto the dialog window blinking the prompt character
<ctrl> D	Assuming you have entered a diagnostic window, returns you to the dialog window.
c	While inside a diagnostic window, changes the context of names and their values so you can investigate the cause of an error.
<shift> <esc>	Creates a sub-dialog window within the current dialog window, suspending the operation of the current window until you type <ctrl> D. Within the sub-window you can type any Smalltalk message.
fix <number>	Enters the Smalltalk editor for a command line in the dialog window. The line is <number> transactions back from where you are currently typing.
redo <number>	Re-sends Smalltalk the message on command line <number> where the line is <number> transactions back from where youare currently typing.

Boxes: An Introduction to Smalltalk

First get the *box* programs by typing:

filin 'boxes' !

After a few blinks they will arrive.

Type to Smalltalk:

☞*joe* ← *box* !

A small box will appear in the top center of your screen. You have given it the name *joe*. As a member of a *class* or group of objects resembling boxes, it can receive messages having to do with "boxness", particularly those concerned with position, size, and tilt. Try:

joe grow 50 !

joe will get bigger. Try:

joe turn 30 !

and

joe grow -20 !

and

joe is ? !

joe will turn, grow, and answer that he is a *box* correctly. Now try:

☞*jill* ← *box* !

A new *box* will appear. Type similar messages to *jill* using different numbers for *size* and *tilt*. *jill* will answer the question *jill is ?* with *box* (as did *joe*).

Now try:

repeat (joe turn 20. jill turn -11) !

Both of the individuals respond. To "escape" from the endless loop, press the key marked 'ESC' located in the upper left hand corner of your keyboard. Ask the questions:

joe's size !

and

jill's size !

(Don't forget that 's is typed as ⟨*ctrl*⟩ *s*)

We see from this and the little "movie" which we created that *joe* and *jill* are really separate entities which can do similar things.

An analogy to these ideas is the common notion of classification by similar properties. For example, we find useful the idea of grouping human beings into a class because we see so many similarities between individuals that we would like to discuss them in the abstract. The class "human" has properties such as 'name', 'age', 'weight', 'walk', 'eat', 'eyecolor', and many others. Each individual human (we often say *instance of the class human*) has particular values for these properties. Some of the values are quantities (as with a value for 'weight'), and some are actions (an individual may have a particular kind of rolling gait for 'walk'). Smalltalk's semantics are at a more comprehensive level than natural language and thus make no distinction between the rather crude English notions of "thing" and "action".

In Smalltalk, every entity is called an *object*; every object belongs to a *class* (which is also an object). Objects can remember things about themselves and can communicate with each other by sending and receiving messages. The class handles this communication for every object which belongs to it; it receives messages and possibly produces a reply, typically a message to send to another object.

The central idea in writing Smalltalk programs, then, is to define classes which handle communication among objects in the created environment. A message is sent to an object by first mentioning the object's name and then mentioning the message. Either the programmer (via direct keyboard typing) or an action that is a reply from a class sends the message.

A Look at the Class Box

As an example of a class definition, here is a *box*. Its various parts are described below. They serve to introduce the special Smalltalk symbols and syntax. It is a very simple class definition, but encorporates most of what is complex about Smalltalk.

Note that you can also examine the classes we have already discussed (*turtle, window*), as well as any that will be introduced, by typing

show ‹classname› !

The definition of *box* is

```
to box var | x y size tilt

    (◀draw      ⇒   (☺ place x y turn tilt. square size.)

     ◀undraw    ⇒   (☺ white. SELF draw. ☺ black)

     ◀turn      ⇒   (SELF undraw. ☞tilt ← tilt + :. SELF draw.)

     ◀grow      ⇒   (SELF undraw. ☞size ← size + :. SELF draw.)

     isnew      ⇒   (☞x ← ☞y ← 256. ☞size ← 50.
                     ☞tilt ← 0. SELF draw)) !

addto turtle ☞(◀ place ⇒ (SELF penup goto (:)(:) pendn up. ⇑SELF)) !

to square length
  (☞ length ← :.
   do 4 (☺ go length turn 90))!
```

addto. The object *addto* is useful for extending the definition of an object (in this case, we used it to extend the definition of *turtle*). Here, we give a *turtle* the ability to respond to the message *place.* The response is to have the *turtle* pick up its pen, go to a screen position that is received as a message, put its pen down, and face in an upward direction (that is, it places itself at a new position without leaving a trace).

square. To draw a square box on the screen, we use the definition of *square* that was constructed in Chapter I. The initial explanation of the colon, :, the Smalltalk symbol for fetching the next value in the message, was also given in Chapter I.

Explanation of the Definition of the Box Class

The format for teaching Smalltalk about a new class of objects is

to ⟨class-name⟩ ⟨temporary variables⟩ /
⟨names of properties describing each member (instance variables)⟩ /
⟨names of properties describing the class (class variables)⟩
(messages to receive and actions to take) !

We use the symbol, *to*, to refer to the next object as a literal class name (here, the name is *box*). Everything following the name is its value; it is useful to think of this format as the mechanism for storing a name with its meaning in a dictionary. There can be different dictionaries for the different contexts in which a message might be sent; typically dictionaries are nested so that an object can gain access to objects and their meanings that were defined in any higher level of context. So far we have only been working at the highest level (top level) of Smalltalk context. The definitions of *box, turtle, window, fontchar,* are found in the top-level dictionary.

Notice that more consistently, we might have preferred the format

☞ *⟨classname⟩ ← class ⟨temporary variables⟩ / ⟨instance variables⟩ / ⟨class variables⟩*
(messages and responses)!

which is more like

☞ *⟨name⟩ ← ⟨value⟩*!

the method for creating instances of the classes. Here we use the symbol ☞ to refer to the next object as a literal name and the part after the arrow, ←, is the object's meaning.

Title Line

Words between the word *to* and the first left parenthesis are referred to as the *title* of the definition. The vertical bar, /, in the *title* is used as a delimiter for the different kinds of variables.

Class and Instance Variables

In the title line, three different kinds of names can be specified: names for temporary storage locations needed only when a member of the class is actually doing something; names of properties that distinguish each member of the class; and names of objects that are common to all members of the class.

The definition of the class *box* specifies two kinds of names: the four properties (*x, y, size, tilt*) that distinguish members of the class; and a temporary variable (*var*). Properties *x* and *y* define the location of the box on the screen; *size* is the length of each of its sides; and *tilt* is its angle of orientation on the screen. Hence, two members of the class *box* can have different screen locations, different sizes, and different orientations.

Messages and Actions

All members of the class *box* respond to messages to grow, turn, draw and undraw. Each member also responds to two messages which have been adopted as reasonable conventions for Smalltalk classes: a request to learn the class type (*is*), and a request to learn about the class' properties (*'s*). The messages that each member of the class can receive, and the actions each will take upon receiving a message, are given within parentheses after the title line.

◀

The symbol ◀, ("eyeball") is in front of each of the message words. The symbol resembles an eyeball because it is used to *look at* the message. Suppose we have created the *box* named *joe* and we send Smalltalk the message

joe grow 100 !

Smalltalk sees the name *joe*, looks *joe* up in its dictionary of names and their associated meanings, and finds that it is an instance of the class *box*. Therefore, Smalltalk runs the definition of the class *box* in the context of *joe*; that is, with the knowledge of a dictionary containing *joe*'s size, tilt, and screen position. For example, *joe*'s dictionary might indicate that size is 50, tilt 0, and x and y coordinates equal to 256.

In sequential order, *joe* looks (with the ◀) for the message *draw*, *undraw*, and *turn*, and then matches the message *grow* with the word *grow* in the definition. Use of the eyeball, ◀, is asking a question: do I see the following token as the next token in the message? We will use the word "token" to refer to a single word or a group of words enclosed by parentheses. Examples of tokens are: *grow*, *(grow 50)*, *read*, *(read eval print)*.

Conditional Actions

Within the main set of parentheses for the class definition, we provide (virtually in tabular form) an itemization of the messages each member of the class can receive and the methods for responding to the messages. This itemization is actually in the form of a conditional statement *(if-clause ⇒ (then-clause) else-clause)*. The *then-clause* consists of the actions that will occur if the *if-clause* has a *not-false* value; it must be enclosed within parentheses.

In the *box* definition, the if-clauses of most of the conditional statements are simply questions "do you see the following word in the message?" Any question that can be answered "false" or "not-false" may be asked in a conditional statement. The choice of the word "not-false" rather than "true" has significance in Smalltalk--any object with a value other than the boolean value "false" is considered to have the boolean value "true". The object, however, returns its "not-false" value for use by the message sender.

The Message Grow

Suppose a *box* sees the message *grow*. The action the *box* takes is to send itself the message *undraw* in order to erase itself from the screen. It then changes the value of *size* by some amount. The specific value of the change is received as a message using the Smalltalk symbol colon, :. In this case, *joe's size* increases by 100. The box then sends itself the message *draw* in order to show itself again on the screen.

The Message Turn

The action taken if a box sees the message *turn* is similar: the box tells itself to undraw, changes the value of the instance variable *tilt*, and then tells itself to draw again.

The Message Draw

The meaning of *draw* is to place the turtle at the box's screen location (x, y), turn the turtle in the box's orientation (*tilt*), and call on the object *square* with the message *size*, the length of each of the box's sides. *Undraw* simply changes the turtle's color to white (assuming the background color is white) in order to "erase" the drawing of the square box.

Again, notice that the evaluation (reading) of a Smalltalk message is done in a left-to-right (linear) manner. As each object is evaluated, it is given the opportunity to read as much of the remaining message as it is able.

The Message Isnew

The Smalltalk object *isnew* is a special question that determines if a new instance of the class is being created. If so, the usual consequent is the action of giving values to each of the instance variables (i.e., describing the new member of the class by assigning values to each name in a dictionary created for the class member). In *box*, the new instance also sends itself a message to draw a square shape on the screen.

> If a Smalltalk class is to have any members (instances) at all, the question *isnew* must be asked as part of the definition of the class.

The Message Move

To have a *box* grow, we change the instance variable *size*; to have a *box* turn, we change *tilt*. To put a box in a different position on the display screen, we want to redraw the *box* with new values for x and y.

Edit *box* and add to the definition

move ⇒ (SELF undraw. x←:. y←:. SELF draw.)

Try

joe move 100 200.

joe move 200 100.!

for i ← 50 to 250 by 10 (joe move i i) !

The third message causes *joe* to move across the screen diagonally from the upper left corner to the lower right corner. To have *joe* track the mouse cursor, simply type

repeat (joe move mx my) !

The above is a method for having the box move to an absolute location on the screen. The *box's* action is to tell itself to erase from the screen (*undraw*), change the values of x and y by receiving new values from the message, and then drawing itself again (*draw*).

Suppose, instead, we would like to type messages such as

joe move right 50. joe move left 100. joe move up 30. joe move down 10.!

In other words, if a *box* sees the message *move*, then it should look for one of the four messages *right*, *left*, *up*, or *down* and then receive a number value to determine by how much to increment x or y. The Smalltalk statement might be

```
move ⇒ (SELF undraw.
          (right ⇒ (x←x+:.)
           left ⇒ (x←x-:.)
           up ⇒ (y←y-:.)
           down ⇒ (y←y+:))
         SELF draw)
```

The use of parentheses around the conditional statement *(right ⇒ ()...)* allows each possible form to evaluate the last part of the statement (*SELF draw*); the reply to the message *move* contains three actions: (1) SELF undraw. (2) look for one of the directional messages, and (3) SELF draw. Also note that moving *up* means decreasing the y coordinate. If we wanted to have both kinds of move methods (relative and absolute) available, we could make one (say the absolute one) the default case. Try

```
move ⇒ (SELF undraw.
          (right ⇒ (x←x+:.)
           left ⇒ (x←x-:.)
           up ⇒ (y←y-:.)
           down ⇒ (y←y+:.)
           x ← :. y ← :.)
         SELF draw)
```

The Message Is.

There are two messages we include, by convention, in each class definition. One is the ability to learn the name of the class; the other is the ability to evaluate messages within the context of the class or class instance. We adopt the word *is* for the first message, and the possessive for 's for the second. If they have not already been included in your definition of *box*, then type

```
addto box (is ⇒ (box ⇒ (↑ box) ? ⇒ (↑ box) 8. ↑ false)
           's ⇒ ( var ← 8.  ← ⇒ (↑ var ← :.) ↑ var eval))!
```

The message *is*, by convention, is a request to learn the name of the class or to ask if the name is the same as one already known. So we might say

	joe is ?!	and be told box
or		
	joe is box!	and be told box (i.e., not-false)
or		
	joe is turtle!	and be told false

The method for responding to *is* (shown in the above definition of *box*) involves seeing () if the class name (in this case, *box*), is the next word in the message. If it is, return (↑) the literal class name (*box*). Otherwise, see if the next word in the message is a question mark (?). If it is, return the literal class name. Otherwise, the answer must be false. In order to not leave the incorrect name sitting in the message, gather it up but do not evaluate it (8). Then return *false*.

The "open colon" symbol (8) is a Smalltalk symbol that says: fetch the next token (the next word or the next words enclosed in parentheses) literally as it appears in the message. The 8 is similar to in looking at the message literally. However, the 8 always fetches in the next literal expression; the only fetches the expression if there is an exact match.

The Message 's

The message ('s) is, by convention, a request to evaluate the next token in the message within the context of the message receiver (typically, the class or the instance of the class). Suppose the size of the box *joe* is 50 and we say

```
☞x ← 100!
☞h ← joe's x!
```

What will be the value of *h*? At the main (top) level of Smalltalk we examine the global dictionary and see that the value of *x* is 100; but, within the context of *joe* (looking in the dictionary created for the class instance), the value is 50. Hence the assigned value of *h* must be 50.

The method for responding to 's involves receiving the next token literally (8), assigning this token as the meaning of a temporary object (here named *var*), and then seeing if the next word in the message is the back arrow (←). If it is a back arrow, then return (⇑) the result of letting the meaning of *var* take on the next value in the message (:). (I.e., this is a method of indirect reference.) If the next word is not the back arrow, then simply return the value of the meaning of *var* (obtained by sending *var* the message *eval*). Again, note that the evaluation of a Smalltalk message is carried out sequentially left to right, but that the message is actually grouped in a right-associative manner because of the Smalltalk method for letting each object read as much of the message as it chooses.

Receiving Messages

There is not one global message to which all message "fetches" (use of the Smalltalk symbols eyeball, ☞, colon, :, and open colon, 8) refer; rather, messages form a hierarchy which we explain in the following way-- suppose I just received a message; I read part of it and decide I should send my friend a message; I wait until my friend reads his message (the one I sent him, not the one I received); when he finishes reading his message, I return to reading my message. I can choose to let my friend read the rest of my message, but then I can not get the message back to read it myself (note, however, that this can be done using the Smalltalk object *apply* which will be discussed later). I can also choose to include permission in my message to my friend to ask me to fetch some information from my message and to give that information to him (accomplished by including ☞, :, or 8 in the message to the friend). However, anything my friend fetches, I can no longer have. In other words,

(1) An object (let's call it the CALLER) can send a message to another object (the RECEIVER) by simply mentioning the RECEIVER's name followed by the message.

(2) The action of message sending forms a stack of messages; the last message sent is put on the top.

(3) Each attempt to receive information typically means looking at the message on the top of the stack.

(4) The RECEIVER uses the eyeball, ☞, the colon, :, and the open colon, 8, to receive information from the message at the top of the stack.

(5) When the RECEIVER completes his actions, the message at the top of the stack is removed and the ability to send and receive messages returns to the CALLER. The RECEIVER may return a value to be used by the CALLER.

(6) This sequence of sending and receiving messages, viewed here as a process of stacking messages, means that each message on the stack has a CALLER (message sender) and RECEIVER (message receiver). Each time the RECEIVER is finished, his message is removed from the stack and the CALLER becomes the current RECEIVER. The now current RECEIVER can continue reading any information remaining in his message.

(7) Initially, the RECEIVER is the first object in the message typed by the programmer, who is the CALLER.

(8) If the RECEIVER's message contains a eyeball, 👁, colon, :, or open colon, ⁸, he can obtain further information from the CALLER's message. Any information successfully obtained by the RECEIVER is no longer available to the CALLER.

(9) By calling on the object *apply*, the CALLER can give the RECEIVER the right to see all of the CALLER's remaining message. The CALLER can no longer get information that is read by the RECEIVER; he can, however, read anything that remains after the RECEIVER completes its actions.

(10) There are two further special Smalltalk symbols useful in sending and receiving messages. One is the keyhole, ⍰, that lets the RECEIVER "peek" at the message. It is the same as the ⁸ except it does not remove the information from the message. The second symbol is the hash mark, #, placed in the message in order to send a reference to the next token rather than the token itself. An example of the use of # is given at the end of the next chapter.

Alternative Box Definition

An alternative method for defining the class *box* is given below. The main difference is the use of the message *redraw* to simplify methods for growing, turning, and moving boxes.

Let's examine the definition in terms of steps (1)-(8) of the previous section. Suppose a box receives a message, message A. In the definition of *box* provided below, if message A contains the token *grow*, the box becomes a CALLER, sending itself another message, B--*redraw* ☞*size←size+:*. The RECEIVER of message B sees the token *redraw*; as a result, it sends itself the message *undraw*. After the action of undrawing is completed, the RECEIVER requests a fetch for a value (:.). The fetch comes from the remaining part of message B (☞*size ← size + :*.). This part of message B contains a colon (:) directing it to get information from the remaining part of the CALLER's message A (as stated in (8) above). This remaining part of message A contains a number that determines the amount of the box's growth. The RECEIVER then sends itself the message *draw*, after which it returns control to its CALLER. The CALLER's actions are now completed.

Similarly for messages containing the tokens *turn* or *move*. In order to change more than one instance variable (that is, both *x* and *y* in the case of *move*), it was necessary to enclose the appropriate messages within parentheses. (Then the fetch for a value found in the action taken by *redraw*, will obtain the value of changing both the *x* and the *y*.) In general, a colon will activate (start determining the value of the message) at the next token--either a single word or words enclosed by parentheses.

The alternative box definition follows.

```
to box var | x y size tilt

  (◁draw      ⇒   (◎ place x y turn tilt. square size.)

   ◁undraw    ⇒   (◎ white. SELF draw. ◎ black)

   ◁redraw    ⇒   (SELF undraw. :. SELF draw.)

   ◁turn      ⇒   (SELF redraw ☞tilt ← tilt + :. )

   ◁grow      ⇒   (SELF redraw  ☞size ← size + :.)

   ◁move      ⇒   (SELF redraw (☞x ← :. ☞y←:.))

   ◁'s        ⇒   (☞var ← 8.  ◁ ← ⇒ (⇑ var ← :.) ⇑ var eval)

   ◁is        ⇒   (◁box ⇒ (⇑☞box) ◁ ? ⇒ (⇑☞box) 8. ⇑false)

   isnew      ⇒   (☞x ← ☞y ← 256.  ☞size ← 50.
                   ☞tilt ← 0. SELF draw)) !
```

Extending the Box Definition. There are several ways to extend or modify the *box* class. We will show one in the next section: the class of polygons, and, after introducing the class *turtle*, we modify the *box* class to be a class whose members each own an instance of the *turtle* class.

Class of Polygons

This simple extension to class *box* allows us to create objects that have any number of sides of equal length. The object that draws any polygon must ask the turtle to draw the appropriate number of lines. After drawing each line, the turtle has to turn enough units so that, after drawing all the lines, the turtle will have turned a complete circle (360 units). Since each angle of a polygon is equal, each turn is an even division of 360 (360/number-of-sides). A polygon-drawing routine is

```
to poly sides size
  (☞sides ← :.  ☞size ← :.
   do sides (☺ go size turn 360/sides))!
```

Using the *box* definition as a model, we can define a class for polygons.

```
to polygon var | x y size tilt sides
```

The title line is similar to that of box; we added the number of sides as an instance variable.

```
(☙ draw ⇒      (☺ place x y turn tilt. poly sides size)
```

The method for drawing has changed. We use poly, not square. poly expects two messages: number of sides and length of each side.

```
☙ grow ⇒       (☙sides ⇒ (SELF redraw ☞sides←sides+:.)
                ☙ size ⇒ (SELF redraw ☞size←size+:.))
```

We adopt message forms

```
joe grow size 100.
joe grow sides 50.
```

as the two alternative meanings of grow. Another method to use is

```
(☞var ← 8.
  SELF redraw var←var eval+:.).
```

Responses to messages *redraw*, *undraw*, *turn*, *'s*, and *move*, are the same as in *box*. The message *is*, by convention, is similar, but looks for the word *polygon*. Or, alternatively, we can take advantage of a Smalltalk object, *ISIT*, and use

```
☙is ⇒ (ISIT eval)
```

This object is part of the basic Smalltalk system referenced in subsequent sections. It is always possible to type *show ⟨class-name⟩* in order to see any such "basic" objects.

In *isnew*, we must give *sides* a value as well as the other instance properties. Suppose we choose to send the initial value of *sides* as a message when we create an instance of *polygon*. I.e.,

```
☞joe ← polygon 3!      creates a triangle
☞joe ← polygon 6!      creates a hexagon
```

Then we write as part of the definition of *poly*

```
isnew ⇒ (☞sides←:. ☞size ← 50.
         ☞tilt ← 0. ☞x←☞y←256.
         SELF draw.)
```

Turtles

The turtle examples in the first section showed some of the messages any turtle can understand. We can get a turtle to draw designs, sketch, and make diagrams with a number of useful and simple programs.

Type

☞*pokey ← turtle* !

Now *pokey* understands messages

go ⟨n⟩	Where n is an integer, move n units forward (+) or backward (-).
turn ⟨n⟩	Where n is an integer, change orientation right (+) or left (-).
penup, pendn	Change state of pen that can leave a trace.
black, white	A turtle can have three ink colors: black, white, or xor.
xor	This color says that whatever "color" is on the screen, show its complement (white for black, black for white). This works only when the turtle's width is 1.
goto ⟨n⟩ ⟨m⟩	where n and m are the horizontal, vertical locations on the display screen.
goto ⟨point⟩	⟨point⟩ is an instance of the class point explained in a subsequent section; try goto mp i.e., goto the point where the mouse cursor is placed.
up	Points the turtle's orientation (dir) towards top of screen.
erase	Clears the window frame in which the turtle lives; default window is the entire screen.
home	Goes to center of the window frame.
← ⟨string⟩	Prints the text (string of characters enclosed by single quote marks) at the turtle's current location, with its direction, width, and color. Note that you can make non-destructive text by using xor ink which complements the background so that reshowing the text erases it while restoring what was underneath.

We can query the *turtle's* property values using 's (typed by striking the key marked 'S' while holding down the 'CTRL' key). For example,

pokey 's *ink*
pokey 's *dir*
pokey 's *width*

Also, *x*, *y*, *pen*, and *frame*. We can change these values by typing

pokey 's *⟨property⟩ ← ⟨value⟩*!

Usually, only the width, whose value is an integer between 1 and 8, and frame, whose value is a display screen window, are modified in this manner. There are alternative methods for each of the other properties.

```
pokey 's width ← 2!
```

A simple design program might be: pokey go a little, turn some amount, go a little more, and so on.

```
to design  var i
      (☞var←:. for i to 300 (pokey go i turn var)) !
```

Try

```
pokey erase home up.
design 89.
pokey home up.
design 91!
```

It is probably better Smalltalk programming style to modify the turtle class definition and give turtles the ability to receive the message *design*. In this way, all turtles, not just *pokey*, will be able to draw designs. *addto* lets us add new messages and responses to class definitions. Try

```
addto turtle ☞(◁design ⇒ (☞var ← :. for i to 300 (SELF go i turn var)))!
```

The explanations of ◁ (eyeball), *SELF*, and ⇒ were given in the previous section. Recall that ◁ is a method for looking at the message and seeing if there is a match between the next word in the message and the word following the ◁. The use of ◁ is a test whose value is either not-false or false. The arrow (⇒) denotes a conditional statement of the form

```
<test for truth> ⇒ (<action to take if the value of the boolean expression is true>)
<otherwise do this>
```

Boxes Owning Turtles

The definition of *box* as presented earlier depends on the turtle to draw each instance of the class. Each time an instance is drawn or erased, must be placed at the appropriate location facing in the appropriate direction. Rather than having to reposition each time, we might assign a turtle to each instance of *box*; since the instance "owns" its turtle, we can assume that the turtle is always correctly positioned.

In the new definition of *box* given below, we use a different *turtle* to draw each instance of the class *box*. The turtle, whom we named *turt*, is an instance variable of the class *box*. Each time we move or turn a *box*, we actually move or turn the *turt* belonging to that *box* . When we draw a *box*, we assume that *turt* is sitting at the correct display coordinate, turned in the proper direction, waiting to draw the geometric shape. The *turt* remembers its position (x, y) and its orientation (tilt) on the screen, so the *box* no longer has to retain this information. There are now only two instance variables: *turt* and *size*.

```
to box var | turt size

(isnew ⇒   (turt ← turtle. size ← 50.
             turt place 256 256.
             SELF draw.)
```

Create turt as an instance of class turtle and give size the value 50. Place the turtle at the starting position and orientation.

```
draw ⇒     (do 4 (turt go size turn 90))
```

Ask the turtle to draw a square.

```
undraw ⇒   (turt white. SELF draw. turt black)
```

Change turtle's ink, assume background is white.

```
redraw ⇒   (SELF undraw. :. SELF draw.)

turn ⇒     (SELF redraw turt turn :.)
```

Rather than changing value of tilt, we simply tell the turtle to change his orientation.

```
move ⇒     (SELF redraw turt penup go (:) pendn)
```

This is a new kind of move--move forward if amount is positive, move backward if negative. Turtle always moves in the direction of his tilt. This is useful if you think of the box as a spaceship!

```
grow ⇒     (SELF redraw size←size+:.)!
```

There were several changes to the *box* definition.

(1) *draw*--we no longer need to reposition the turtle because *turt* is already correctly positioned, nor do we need to use the object *square*.

(2) *turn*--since the turtle must sit in the proper direction, we tilt the box by changing the turtle's direction (send *turt* the message *turn*). The box no longer has instance variable *tilt*.

(3) *move*--the turtle remembers his, and therefore the box's, position. The box no longer has instance variables *x* and *y*.

Dispframes: An Introduction to Text Display

Smalltalk *dialog* windows are instances of the basic Smalltalk system class *dispframe*. Members of this class can show text in a rectangular area that can be framed with thick black lines. As you have already seen, Smalltalk can have many *dispframes*, each one capable of moving its screen position, changing its size, displaying text, and hiding itself (deleting its representation from the display screen). To do these tasks, an instance of *dispframe* understands messages such as *moveto ⟨upper left corner x⟩ ⟨upper left corner y⟩, growto ⟨lower right corner x⟩ ⟨lower right corner y⟩, show, display,* and *hide*. You have sent messages to the windows by pointing at one of the four corners. To help in this task, a *dispframe* understands the messages *hasmouse*, to determine whether or not the mouse cursor is inside the window; and *corner ⟨x⟩ ⟨y⟩*, to determine at which corner, if any, the mouse cursor points. The response to the message *corner* is a number between 1 and 4 depending on the display coordinates x,y.

Each instance of a *dispframe* remembers text that is displayed in the rectangular area. This text is named *buf*. One of the jobs of the class *dispframe* is to fit the text into the window:

(1) changing physical lines when the characters fill the line space ("line wrap around"),

(2) lining the characters up evenly in the right margin (right justify),

(3) scrolling (deleting the initial characters and readjusting the remaining characters upward) when the window can not properly contain all the text.

Placing Text on the Display Screen

There are three ways to place text on the display screen, one uses a turtle, the other two rely on the class *dispframe*.

With Turtles.

```
amy ← turtle!
amy penup goto 100 100 pendn!          Amy has width = 1 and faces upward.
amy ← 'hello'!                         Note the need for single quote marks as delimiters.
```

The word "hello" appears on the screen. The upper left corner of the first character shows at *amy*'s x,y position. Now *amy* has been repositioned at the end of the displayed word.

```
amy's width ← 2!                       Increase amy's width to 2.
amy ← 'hi'!                            Print another word.
```

Try printing with turtles facing in different directions and having different widths and colors. Although it is possible to print text on the display with a turtle facing in any direction, text generally looks best when the turtle's direction is horizontal, vertical, or at 45 degree angles.

With Display Frames. To create a *dispframe* you send at least five messages describing the rectangular area and its contents: the area's upper left corner x, its width, its upper left corner y, its length, and a string. The string is the method for storing the text characters to be displayed.

```
dp ← dispframe 100 75 100 120 string 200.!
```

This creates a rectangular area 75 x 120 at location 100,100. It can contain up to 200 text characters. The simplest way to place text in this area is to send the *dispframe* the message *put*.

```
⟨dispframe⟩ put ⟨text⟩ at ⟨x⟩ ⟨y⟩!
```

Where x,y are the display screen coordinates. For example,

dp put 'hi there' at 150 100!

Now try

dp put 'hi where' at 200 150!

Notice that the *dispframe* has changed its x,y position to 200,150. It has replaced its original text with the text *'hi where'*, but it has not erased the original text *'hi there'*. Try

repeat (dp put 'hi' at mx my)!

to place the word "hi" all over the screen.

Appending Text to Display Frames. A *dispframe* stores its text in a place named *buf*. The message ←, when sent to a *dispframe*, is a request to add characters to *buf*; *buf* is an instance of a basic class named *string*. We can print the word "hello" in the *dispframe dp* by typing:

dp ← 'hello'.!

Now try:

dp ← 'how are you today? My name is dp and I am a dispframe'!

Do you see how the line-wrap-around works? And that spaces have to be explicitly stored into the dispframe? The original text was not cleared when new characters were added; rather, the new characters are appended to the end. Now try the various other messages to a *dispframe*:

dp hide! *dp display*!	The entire area disappears and reappears.
dp fclear! *dp show*!	The text area is cleared and represented.
dp clear! *dp show*!	This empties the string buf so there is no longer text to display.
dp hide! *dp growto 250 250*! *dp display*!	Now the frame is larger.
dp hide! *dp moveto 50 50*! *dp display*!	Now the frame is in a new position.

Boxes as Menus

The Smalltalk class editor uses two instances of *dispframe.* The first is the window containing the levels of the class definition; the second is the *menu* window. In each case, you were able to position the mouse cursor in the window and the editor was able to determine which character or word you were grabbing. Instances of *dispframe* understand three messages that aid in this task:

mfindc (which character),
mfindw (which word), and
mfindt (which token, that is, which word
or set of words enclosed in parentheses).

The next example was chosen in order to clarify the use of these messages and to provide an example of a *dispframe.*

A *menu* is an ordered list of objects that can be selected in a variety of ways. One way is to point at the object with the mouse cursor. The objects might be words or pictures, each representing things to do, or names of other objects to retrieve or to "activate" (that is, give the ability to do something, such as to receive and/or to send messages).

We have chosen a simple example of a menu consisting of a list of words, each word being the name of a polygon. The result of grabbing a word will be to create the corresponding instance of the class *polygon.* Before the new instance is actually created, the user will select the position on the screen where the polygon is to be drawn.

We will use a modified version of the definition of *polygon,* one in which the polygon position is determined from a message received at the time the object is created. For example, we will create the polygon *joe* by typing

```
joe ← polygon 5 150 100!                    joe is a pentagon (5 sides) at 150,100

to polygon | sides size                     polygon simply creates the object.

    (draw⇒ (do sides (go size turn 360/sides))
                                            Draws it on the screen.
    isnew ⇒ (sides ← :. size ← 50.          Values for sides and the turtle's position are
             ← turtle. place (:)(:).        provided when the polygon is created.
             SELF draw))!
```

The definition of *polygonmenu* includes the instance variable *codevector.* This object will be an instance of the basic Smalltalk class *vector,* a method for storing a list of things. In this case, we store a list of the names of the possible polygons to create. For example, we might create a menu by typing:

```
pm ← polygonmenu (triangle square pentagon hexagon septagon octagon)!
```

The list *codevector* owned by *pm* is now a list of polygon names that will appear in the menu box on the screen. Each item in *codevector* refers to a polygon that can be created.

```
to polygonmenu i | dp codevector

    (isnew ⇒(☞codevector ← 8.
```

When creating a menu, fetch literally the vector of words to be displayed in the menu.

```
        repeat (button 4 ⇒
```

Wait for the user to press button 4 to indicate the menu position; then create dp, the dispframe, at the mouse cursor's position;

```
            (☞dp ← dispframe mx 75 my 120 string 100.
```

and print each word in the menu followed by a carriage return.

```
            for i to codevector length - 1

                (dp ← codevector[i] chars. dp ← 13).

            done))))!
```

We reference items in a vector using the notation: name[index]

The above definition of *polygonmenu* simply shows a rectangular area filled with words. The method for printing each word from the list is to count down through each item using the *for* iteration method. The counter is *i*; *codevector[i]* refers to the ith item. For example, in the above, if i=1 then *codevector[i] = codevector[1] = ☞triangle.*

Each item in the list is an *atom*, a basic Smalltalk system class. Each instance of an atom responds to the message *chars* by forming a string of characters for the atom value. For example, the response from the atom *☞triangle* would be the string *'triangle'*. The word "triangle" is printed in a dispframe area by sending the string *'triangle'* to the dispframe. Hence the contents of the *for* iteration is to send the dispframe *dp* the string *codevector[i] chars.*

The code for a carriage return is 13. Hence *dp←13* is a method for printing a carriage return in the dispframe. This causes each new word to appear on a new line in the menu.

Now let's find the word to which the mouse cursor points.

```
addto polygonmenu ☞(☜index ⇒ (⇑dp mfindt mx my))!
```

If we send a *polygonmenu* the message *index*, we will receive a list (vector) of four numbers (the reply from the dispframe). The four numbers are: the actual index of the word in the vector *codevector*, the x position of the first character in the word, the width of the word, and the y position of the first character in the word. Suppose, as an example, we type *pm index* while we are pointing to the first word in the menu.

```
pm index!
 (1 65 50 100)
```

The result is a vector. The first number in the vector is the index of the word in the menu. The second is the x position, third the word width, and fourth is the y position. Word height is generally 14.

To select the menu word from *codevector*, we retrieve the i[1]th item in the vector.

```
addto polygonmenu ☞(☜select ⇒ (☞i←SELF index. do something with codevector[i[1]]))!
```

Suppose we want to delay computing *i* until the user can point into the menu and press a mouse button.

```
select ⇒ (repeat (button 4 ⇒ (i ← SELF index.
                              do something with codevector[i[1]]. done)))
```

The *done* part is important. It stops the *repeating* and returns control to the message sender. What we do is simply to call on the *polygon* class with sides = 2+i[1]. Hence, in this case, it is actually not necessary to retrieve the i[1]th item in *codevector*.

```
select ⇒ (repeat (button 4 ⇒ (i ← SELF index.
                              polygon 2+i[1] mx my. done)))
```

But, again, there is no delay provided in order to allow the user to point someplace on the screen before the figure is drawn. Let's change the response to *draw*.

```
select ⇒ (repeat (button 4 ⇒ (i ← SELF index.
                              SELF draw 2+i[1]. done)))

draw ⇒ (repeat (button 0 ⇒ (done))
          repeat (button 4 ⇒
                    (polygon (:) mx my. done)))
```

Make certain that the button is released. Then wait for button press before calling on polygon.

We can complete the menu selection by adding the ability to complement the color of the selected word. There is a special routine, *dcomp*, that lets us complement any rectangular area of the screen. It expects four messages: the area's upper left corner x, the width, the upper left corner y, and the height. For example:

```
dcomp 100 50 100 200!
```

Try

```
do 100 (dcomp 100 50 100 200)!
```

The height of the font we are using is 14, so, to complement a word in the menu, we use

```
dcomp i[2] i[3] i[4] 14.
```

The change to the class definition is

```
select ⇒ (repeat (button 4 ⇒
                  (i ← SELF index.
                   dcomp i[2] i[3] i[4] 14.
                   SELF draw 2 + i[1].
                   dcomp i[2] i[3] i[4] 14.
                   done)))
```

Of course, we assumed the index was a reasonable number. It is safer to check! We change the response to *index* to first see if the mouse cursor is inside the frame, and, if so, to compute *i* and check to see if *i = -1*. If it does, then the cursor was inside the frame but was not pointing at any token. The completed definition is:

```
to polygonmenu  i | dp codevector
  (↞index ⇒ (dp hasmouse ⇒(☞i ← dp mfindt mx my. i[1] = -1 ⇒ (⇑ false) ⇑i)
            ⇑false)
    ↞select ⇒ (repeat (button 4 ⇒
                    ((☞i ← SELF index) ⇒(dcomp i[2] i[3] i[4] 14.
                                         SELF draw 2 + i[1].
                                         dcomp i[2] i[3] i[4] 14.done)
                    done)))
    ↞draw ⇒ (repeat (button 0 ⇒ (done))
            repeat (button 4 ⇒ (polygon (:) mx my. done)))
    isnew ⇒ (☞codevector ← 8.
            repeat (button 4 ⇒ (☞dp ← dispframe mx 75 my 120 string 100.
                                for i to codevector length - 1
                                  (dp ← codevector[i] chars. dp ← 13). done))))!
```

Another kind of menu might use the index of the menu word selected to choose a message to evaluate. The message might be an item in a vector of messages. For example, suppose we did not want to depend on the order of the *polygonmenu* to determine which polygon was created. Possibly, we want a menu to be

hexagon
triangle
circle

Within the repeat-loop of the response to the message *select*, replacing *SELF draw 2+i[1]*, we might have

```
☞((polygon 6 mx my) (polygon 3 mx my)(polygon 10 mx my)) [i[1]] eval
```

Here i[1] is the index into the vector of messages. We select an item from the vector and send it the message *eval* in order to obtain the desired polygon.

Chapters IV and V contain more information and examples about the classes *dispframe* and *vector*.

A Few Sketching Tricks

Some of our favorite design programs are presented below. Caution: if you copy these routines, be certain that you have a large enough window to accommodate all your typing. Smalltalk only sees text that you can see in the window. You can type part of the routine and add the rest by using the Smalltalk editor. Alternatively, you can retrieve these turtle routines from the disk pack by typing

```
filin 'turtlefns'!
```

dragon

```
to dragon length
 (length ← :.
  length = 0 ⇒ (go 10)
  length > 0 ⇒ (dragon length -1. turn 90. dragon -(length-1))
  dragon -length+1. turn - 90. dragon length + 1.)!
```

Try

```
erase home up. dragon 8!
```

hilbert space filler

```
to hil i a b
  ((i ← :) = 0 ⇒ (turn 180)
    (i > 0⇒
      (a ← 90. b ← i - 1)
       a ← - 90. b ← i + 1)
   hil1 hil2 hil1)!

to hil1
  (turn a. hil 0 - b. turn a)!

to hil2
  (go 10. hil b. turn 0 - a. go 10 turn 0 - a. hil b. go 10)!
```

i is the recursion number. Try

```
erase home up !
hil 4!
```

squiggles

```
to squig90
 (repeat
  (home do 200
   (go rand / 1000 turn 90 * rand mod 4)))!

to rand (↑i ← i * 5)!
```

Try

```
erase. 's width ← 2. i ← 11. squig90!
```

Or

```
to squiggle i
  (☞i ← 13.
   repeat
    (☺ home.
    do 1000
     (☺ go 10 turn rand)))!

☺ erase. ☺ 's width ← 1. squiggle !
```

Changing ink color and the width of the *turtle*'s trace makes for interesting variations. Try

```
☺ home up erase. ☺ 's width ← 1. dragon 8.
☺ home up turn 90. ☺ 's width ← 2. dragon 8. !
```

Sketching. We can sketch by telling any turtle to follow the mouse cursor. For example,

```
repeat (pokey goto mx my) !
```

or

```
repeat (pokey goto mp) !
```

The routine *mp* returns the point where the mouse is located (that is, it combines mx and my). Members of the class *point* respond to messages *x y + - = max min.* This class is described in more detail in Chapter IV.

More sketching control is obtained with the mouse buttons.

```
to draw
  (repeat
   (button 4 ⇒ (pokey pendn goto mp)
    button 2 ⇒ (pokey erase)
    button 7 ⇒ (done)
    pokey penup goto mp)) !

draw !
```

Now lines are drawn only when you press the top mouse button (button 4); the bottom mouse button (button 2) erases the screen; holding down all the mouse buttons (button 7) terminates the program; otherwise, the *turtle* moves to the cursor without leaving a trace. (Note, there are two versions of the mouse device, one having buttons ordered from top to bottom, the other ordered left (top) to right (bottom). Henceforth, we will refer to the top-to-bottom version.)

Variations use the mouse button to control changing the *turtle*'s width and changing *turtle*'s ink color to allow selective erasure.

"Rubber Bands" is another sketching technique in which a *turtle* expands and contracts straight lines, always stretching towards the mouse cursor. The line starts at the point indicated by pressing the top mouse button; the bottom mouse button indicates that the line is to be fixed in its current position.

```
to rubberband fp sp
  (repeat
   (button 4 ⇒ (🐵 penup goto 👉fp←mp pendn.
                repeat
                 (🐵 goto 👉sp←mp.
                  button 2 ⇒ (done)
                  🐵 white penup goto fp pendn goto sp goto fp black)))) !
```

Saving the points *fp, sp*, lets you store the method for constructing the drawing. A simple example of storing mouse points is

```
👉 points ← stream of vector 10!
repeat (🐵 goto points ← mp)!
```

Here, the object *points* is an instance of the class *stream*, a method for storing other objects (described in detail in Chapter IV). Members of the class *stream* respond to messages ← *contents next reset end*. Each time the turtle moves, the new turtle location is stored (←) in *points*. The routine *rubberband* can be modified to store each pair (*fx, sx*), making these lines available for reconstructing the sketch.

```
to newrubberband fp sp points
   (👉points ← stream of vector 10.
    repeat
     (button 7 ⇒ (done with stream of points contents)
      button 4 ⇒ (🐵 penup goto 👉fp←mp pendn.
                  repeat
                   (🐵 goto 👉sp←mp.
                    button 2 ⇒ (points ← fp. points ← sp. done)
                    🐵 white penup goto fp pendn goto sp goto fp black)))) !

👉 points ← newrubberband!
```

The sketch can be reconstructed by

```
to reconstruct pts
        (👉pts ← :. pts reset.
         repeat (pts end ⇒ (done)
         🐵 penup goto pts next pendn goto pts next))!

reconstruct points!
```

That is, reset the *stream*, and repeatedly retrieve the *next* item until reaching the *end*.

Chinese Brush Strokes. Changing the width of the *turtle*'s path as a line is being drawn leaves "Chinese Brush Strokes". This class lets you draw variable-width lines as long as you press the top mouse button.

```
to brush i 🐵
      (👉🐵 ← turtle.
       repeat (button 2 ⇒ (🐵 erase)
               button 4 ⇒ (🐵 pendn.
                           repeat (🐵's width ← 👉i←1+i mod 8. 🐵 goto mp.
                                   button 0 ⇒ (done)))
               🐵 penup goto mp. 👉i←0.)) !
```

Feather Strokes. This next class varies the thickness of the trace depending on the direction of the "feather stroke".

```
to feder ox oy nx ny
    (☺ penup.
     repeat
      (button 4⇒ (☺ goto ☞ox ← mx ☞oy ← my pendn.
                  repeat
                  (button 0 ⇒ (☺ penup. done)
                  ☺'s width ← 1 + abs (3 * (☞ny ← my) - oy) /(☞nx ← mx) - ox.
                  ☺ goto ☞ox ← nx ☞oy ← ny))
       button 2 ⇒(☺ erase)))!

to abs x
    ((☞x ← :) < 0 ⇒ (↑0 - x) ↑x)!
```

Cobwebs This last class uses a second *turtle*, *turt*, to form cobwebs around the lines drawn by ☺. The creation of this *turtle* with the message *frame* is explained in Chapters IV and V; the class *vector* is also explained in Chapter IV. A *vector* is used here as a method for storing ☺'s display coordinates for use by *turt*. The class *cobweb* expects two messages, the color of ☺'s ink and the color of *turt*'s ink. ☺'s width is set to 3 and *turt*'s width is set to 1. Cobwebs are drawn as long as you press the top mouse button. Clearly, this sketching method is designed for the color version of Smalltalk.

```
to cobweb n i xs ys turt
    (☞n ← 10.  ☺'s width ← 3.  ☺'s ink ← :.
     ☞turt ← turtle frame ☺'s frame.
     turt's width ← 1.  turt's ink ← :.
     ☞xs ← vector n.  ☞ys ← vector n.
     repeat
      (button 4⇒
       (xs[1 to n] ← all mx.  ys[1 to n] ← all my.        store mx in all of vector xs
                                                          store my in all of vector ys
       ☞i ← 1.
        ☺ penup goto xs[1] ys[1] pendn.
        repeat
         (0 = mouse 4⇒(done)
          ☞i ← 1 + i mod n.
          turt penup goto xs[i] ys[i].
          ☺ goto xs[i] ← mx ys[i] ← my.
          turt pendn goto xs[i] ys[i])))!
```

In the black-and-white version of Smalltalk, type

```
cobweb (-3) (-3)!
```

Paint Brush

Smalltalk also has a method for transferring blocks of designs, such as a solid black rectangle, or one specially constructed to resemble a gray "color". The basic method of interfacing brush painting to Smalltalk is through the class *rectangle*. This class definition is available by typing

```
filin 'xyfns'!
```

A sufficient abbreviated version is

```
to rectangle | origin extent
  (◁ has ⇒ (☞t ← :.
            ⇑ origin  t  origin + extent)
    ◁ center ⇒ (⇑ origin + point extent x/2 extent y/2)
    ◁ 's ⇒ (⇑ 8 eval)
    ◁ is ⇒ (ISIT eval)
    ◁ paint ⇒ (CODE 43)
    isnew ⇒ (☞origin ← :. ☞extent ← :.))!
```

As you can see, this definition includes an escape to machine code (CODE) which supports the movement of bits on the display screen. The two instance variables, *origin* and *extent*, must be instances of the class *point*, a basic system class defined completely in Chapter IV. The class *point* is a method for working with two coordinates as one entity, for example, as a display point. To create a *rectangle*, type

```
☞source ← rectangle
        ‹upper left corner point›
        ‹extent of area as a point whose parts are the area's width and height›!
```

For example, try

```
☞source ← rectangle point 50 50 point 10 20!      width is 10, height is 20
```

The rectangle does not, as yet, appear on the display.

Suppose you want to fill the rectangle with "color". "Gray color" is obtained by combining black and white dots to form a spatial half-tone which gives the impression of a gray color (like that in newspaper print). The number 1 represents a black dot, 0 a white dot. The "paint brushing" works by painting "gray" into the source rectangle and then transferring from the source to a destination. The destination is designated as a *point*, the upper left corner of a rectangle that will be made the same size as the source. "Gray" is specified as an integer which gets folded into a 4x4 rectangle to form a pattern which then gets replicated throughout the area being painted. The folding is

```
A B C D  --->
                ----
                | A |
                | B |
                | C |
                | D |
                ----
```

Where A,B,C,D are binary numbers. For example, suppose the desired gray pattern is

1101
0111
1101
0111

The corresponding single binary number is

1101 0111 1101 0111

which in octal is 0153727. Hence, the integer to store as the paint "color" is 0153727. (Note, octal numbers in Smalltalk must begin with the number 0.) Try

dest ← mp.!	Place the mouse cursor somewhere on the screen.
source paint 12 0153727!	Store the gray "color" into the source rectangle.
source paint 0 dest!	Copy the source into the destination.
source paint 0 mp!	Copy the source into the mouse point destination.

Now try

source paint 4 dest!	Copy the complement of the source area into the destination.
source paint 8 dest 32125!	The integer 32125 is another "gray" color. This brushes the new gray into the destination where the destination is a rectangle the same size as the source.

The number following the message *paint* is an operation indicator. As we have seen:

0	copy source to destination point
4	copy complement of source to destination point
8	source brushes a new gray to destination point
12	fill source with a gray

Each of these four operations has one of 4 modes, obtained by adding the following integers to the above operation code.

0	store source into destination (paint--do operation as indicated above)
1	OR source into destination (merge the 1's and 0's)
2	XOR source into destination (invert)
3	AND complement of source into destination (erase)

Hence, you might try the following variations using objects *source* and *dest* defined above.

source paint 1 dest!	Take source and OR it to the destination.
source paint 2 dest!	Take source and XOR it to the destination.
source paint 5 dest!	Take complement of source and OR it to the destination.
source paint 10 dest 32125!	Source brushes the XOR of the gray (32125) to the destination.

and so on. Some integers you might use as gray include (these are decimal numbers)

-1 32125 -5161 -21931 23130 15420 5160 -32126 0 11892 -10213 13260 51 -52

(Recall that the negative indicator sign is typed as *<shift>*-, that is, press the key marked '-' while holding down the key marked 'SHIFT'.)

Suppose you want to create a shaped area of gray color in the upper left portion of the screen.

```
palette ← rectangle point 0 0 point 16 16!
```

The shape can be a paint brush shape.

```
brush ← rectangle point 20 20 point 16 16!
```

and the tone is one of the numbers representing the gray color.

```
tone ← 15420!
```

The palette is then the mixture of brush and tone. Design the brush.

```
penup goto brush center pendn.
's width ← 8.
do 2 ( go 2 turn 90)!
```

The combination is

```
brush paint 8 palette's origin tone!
```

To spread the paint around, try

```
repeat (button 4 ⇒ (palette paint 8 mp tone))!
```

Try building your own painting system using the Smalltalk painting brushes.

BITBLTing. A part of the Smalltalk system is the ability to move blocks of bits (0's and 1's) from one part of the memory of the computer to another, quickly. The Smalltalk program that should be used with caution is

```
to BLT (CODE 41)!
```

It requires twelve messages which are, in order:

1	base address of the destination of blocks of bits
2	destination raster
3	destination x
4	destination width
5	destination y
6	destination height
7	operation code as defined above for *paint*
8	base address of the source of blocks of bits
9	source raster
10	source x
11	source y
12	gray color

Without too much explanation, we offer the following useful definitions for saving and changing the shape and color of the mouse cursor.

```
to cursor p buf gray
    (◀ loadfrom ⇒
        (☞p ← :.
         BLT 281 1 0 16 0 16 0 mem 60 32 p x p y 0)
     ◀ copyto ⇒
        (☞p ← :.
         BLT mem 60 32 p x 16 p y 16 0 281 1 0 0 0)
     ◀ show ⇒
        (☞buf ← :. ☞p ← PNT buf.
         BLT 281 1 0 16 0 16 0 p+2 1 0 0 0)
     ◀ makebuff ⇒
        (☞buf ← string 32.
         ☞ p ← PNT buf.
          BLT p+2 1 0 16 0 16 0 281 1 0 0 0.
          ↑ buf))!

to PNT (mem 255 ← :. ↑ mem 255)!
```

Try

```
☞source ← rectangle point 0 0 point 16 16!
```

```
☞savecursor ← cursor makebuff!
```
A string containing bits representing the cursor.

```
source paint 12 "5161!
```
Paint gray color in the source rectangle.

```
cursor loadfrom source's origin!
```
loadfrom requires a pointer to the upper left corner of a 16x16 area (source rectangle upper left corner).

```
cursor show savecursor!
```
Restore the cursor to original shape.

Or try the palette example given earlier. Then say

```
cursor loadfrom palette's origin!
```

Now

```
repeat (button 4 ⇒ (palette paint 8 mp tone))!
```

The cursor looks like the paint brush!

Chapter III. THE SMALLTALK WORLD AND ITS PRIMITIVES

Up to this point, we have provided a "try it and see the flavour of what happens" style of presentation. In this chapter, and in the next, we modify the style in order to provide a direct discussion of the basic Smalltalk concepts: classes, instances, and message sending and receiving. We assume, however, that the reader has examined earlier chapters and is familiar with the special Smalltalk symbol set presented there. The following is a summary of these symbols.

◀	look to see if a specific word appears as the next word in the message.
:	receive the next value from the message.
⌾	receive the next literal token (single word or words enclosed in parentheses) from the message.
⇒	indicates conditional statement: if-clause ⇒ (then-clause) else-clause.
⇑	return the following object; the object is "active" in the sense that the next action taken is to run this object's class definition and to let this object examine the message.
SELF	name used to refer within a class definition to the active instance of a class.
/	delimiter used between names of class, instance, and temporary variables in the title line of a class definition.

Objects

Every entity in Smalltalk's world is called an **object**. Objects can remember things and communicate with each other by sending and receiving messages. Every object belongs to a class (which is also an object). The class handles all communication (receiving a message and possibly producing a reply) for every object which belongs to it.

Examples of objects:

Class Name	**Objects**
number	*3 4 3.14159 6.28e-23*
string	*'this is some text' 'here is some more'*
atom	*x y file3 number*
vector	*(1 3 5 7 9 11 13)*
turtle	☺

Message Sending and Receiving

A message is sent to an object by first mentioning the object and then mentioning the message.

Messages are simply strings of words separated by spaces. A "word" is either (1) a string of alphanumeric characters beginning with an alphabetic character, (2) a string of all numeric characters, or (3) one of the special symbols listed above, ☺, or any arithmetic operator.

Examples of sending messages:

	Communication	Object	Message	Reply	Graphics Action
1.	*3+4+5*	*3*	*+4+5*	*12*	*none*
2.	*5 mod 3*	*5*	*mod 3*	*2*	*none*
3.	*'abc'+'def'*	*'abc'*	*+'def'*	*'abcdef'*	*none*
4.	*joe go 100*	*joe*	*go 100*	*joe*	*draws a line 100 units long*
5.	*do 4 (joe go 50 turn 90.)*	*do*	*4 (joe go 50 turn 90.)*	*none*	*draws a square with side 50 units long*
6.	*joe grow 50*	*joe*	*grow 50*	*none*	*joe, the box, grows his sides by 50 units*
7.	*joe turn 25.* *jill grow 30.*	*joe* *jill*	*turn 25* *grow 30*	*none* *none*	*joe turns 25 degrees* *jill grows her sides 30 units*

The class of an object can receive messages in a variety of ways. In addition, the user can add new ways for messages to be received. Once a message is received, the object can take some action, such as returning a message to the sender (*reply*) or modifying a graphic display (*graphics action*).

Notes on the Examples:

	Communication	Object	Message	Reply	Graphics Action
1.	3+4+5	3	+4+5	12	none

The expression *3+4+5* is handled by sending the reply of the message *4+5* back to *3*. First, let's look at a simpler message: *3+4*. In the class *number*, we have

+ ⇒ (b ← :. ⇑ 'result of computing the sum of SELF and b')

The action taken after seeing the '+' is to receive a value from the message and give it the name *b*. Then return (⇑) to the sender a reply calculated somehow. The calculation uses the value of the active instance of the class (referred to by the name SELF) as well as the value of *b*. In the simplified example, the value of SELF is 3 and the value of *b* is 4. (This is usually done using more Smalltalk code as in the first example, but can also be an escape to lower levels of the system, as in this example. Such escapes are seen in the definition as CODE <number>.)

Hence, after seeing the '+', the receiver (*3*) receives a value (*4*) and returns the sum (*7*).

In example 1, after the object *3* first sees the message *+*, the action *b ←:.* tries to receive a value from the rest of the message. In this case, the rest of the message is *4+5*. The *4* is a *number* also. It is sent the message *+5*, which will activate the same line in the definition of *number* as *3* was using. *4* sees the *+* and tries to get a value (*5*) into ITS 'b'. There is nothing more in the message so 4+5 is computed and *9* is returned to *3* as the value of its message. The *3* adds itself to the *9* and returns *12* to the original sender. All messages in Smalltalk are handled in a similar manner.

	Communication	Object	Message	Reply	Graphics Action
2.	*5 mod 3*	*5*	*mod 3*	*2*	*none*

In the example above, a message is sent to a member of class *number* (the literal 5). 'mod' is a token which class *number* can recognize (we'll see how in a bit). It indicates a desire for finding the modulo of the number with respect to another number. We need another item from the message, this time a numerical value. The part of class *number* which receives this general message form looks like:

*mod ⇒ (⇑ SELF - (☞ b ← :.) * SELF / b)*

This means: if, in the message,

you see	mod
the word 'mod'	
then	⇒
do the following:	
receive a value from	
the message and give	
it the name *b*	☞b ← :.
then	
return to the sender	⇑
a reply calculated by	
dividing yourself	
by the value received;	SELF/b
multiplying the result	
by that same value;	b * SELF/b
and subtracting this last	
result from yourself.	SELF - b* SELF / b

To clarify the right-associative nature of the evaluation, we add the following, somewhat redundant explanation of the above message. The uparrow (⇑) expresses the action of actively returning some value (that is, the returned value is an object that becomes the immediate next message receiver; it is able to examine the rest of the message). The value returned is obtained by evaluating the next object in the message, here, SELF. Because SELF is an instance of class *number*, it looks for and finds an arithmetic operator (-) and asks to fetch the next value from the message. This in turn effects the evaluation of the parenthesized message *(☞b←:.)*. The value received is a *number*, hence the value of b is an instance of *number*. This instance is still active and is able to look at the message and see the multiplication operator (so far, the subtraction has not been completed). Upon seeing that multiplication is indicated, a fetch is made for the multiplier. This activates the second reference to SELF, a number that sees the division, retrieves the value of b, and completes the division operation. The result of the divison operation is the multiplier; the result of the multiplication is the subtrahend; the result of the subtraction is the value returned.

Most lines in class definitions resemble this one strongly because Smalltalk is modelled on the notion of communication by sending and receiving messages.

Since everything in Smalltalk is an object and every object can send and receive messages, "expressions" (as in example 1) can be built by simply sending more messages to returned values which have already been calculated. The messages can be cascaded in a single message stream, or determined conditionally as actions specified in a class definition. Message streams are typed to Smalltalk by the user or included as part of the definition of a class.

If a *number* can answer the question *is number* affirmatively, then we can easily test the value in the previous example (which was given the name 'b') by:

```
«mod ⇒ ((☞b ← :.) is number ⇒ (⇑ SELF - b * SELF / b)
                    error ☞('non-numeric operand'))
```

We don't usually bother to do this as it is much better for the action to discover that a value is of the wrong class by sending a message which it doesn't understand.

The object *error* handles printing the specified message in a Smalltalk sub-window and letting the user investigate the context of the error.

	Communication	Object	Message	Reply	Graphics Action
3.	'abc'+'def'	'abc'	+'def'	'abcdef'	none

Class *string* has a way very similar to *number* for receiving a message and then doing something. Here, the action is string concatenation.

```
«+ ⇒ (☞b ← :. ⇑ 'result of concatenating SELF and b')
```

In other words, receive a value from the message and give it the name *b*. Then return to the sender a reply calculated somehow. Again, this is probably done using an escape to lower levels of the system.

	Communication	Object	Message	Reply	Graphics Action
4.	😊 go 100	😊	go 100	😊	draw a line 100 units long

The message to the turtle to go 100 units (100 "dots" on the display screen) is received in a manner similar to the second example. A turtle actively returns itself, thus permitting the cascading of turtle messages.

```
«go ⇒ (☞dist ← :. 'Somehow make turtle go dist' ⇑ SELF)
```

	Communication	Object	Message	Reply	Graphics Action
5.	do 4 (😊 go 50 turn 90.)	do	4 (😊 go 50 turn 90.)	none	draws a square with side 50 units long

"Control Structures" in Smalltalk work the same way. The object *do* receives its message:

```
☞N ← :. ☞exp ← 8. 'method for doing exp N times'
```

The 8 means receive the message "literally". We use it here because we don't want the value of 😊 *go 50 turn 90* (which are actions by the *turtle*), but rather its literal form (which is a request for actions by the *turtle*) to be iterated over and over. We do want to calculate a value for the repetition number to allow expressions such as:

```
do a+b*5 (...)
```

	Communication	Object	Message	Reply	Graphics Action
6.	joe grow 50	joe	grow 50	none	joe, the box, grows his sides by 50 units; a larger box is displayed

This is a typical message to a graphical object. We will show both the receipt of the message and its method:

```
grow ⇒ (SELF undraw.
        size ← size + :.
        SELF draw)
```

When *grow* is seen, we 'undraw' ourSELF using the old size, compute the new size by adding a new value received to the old size, and tell ourSELF to 'draw' using the new size.

	Communication	Object	Message	Reply	Graphics Action
7.	joe turn 25. jill grow 30.	joe jill	turn 25 grow 30	none none	one box on screen tilts 25 degrees, and then another box grows 30 units

Here we see a bunch of send messages done in sequence. The period '.' terminates a message and hence separates two message communications. In many cases, the period is not needed, as the message receiver will be able to determine how much of the message to examine. The period does, however, serve the syntactic purpose of disambiguating the end of a message.

The order of communications is done sequentially from left to right (as with English text), so:

joe turn 25.

is done before

jill grow 30.

The Notion of Class

The basic class definition deals with just two ideas:

1. The notion of creating objects which have independent existence and memory.

2. The control of the flow of evaluation by sending and receiving messages in various ways.

For example, a **send** message is a control action because flow of control is suspended in the **sender** and resumed in the **receiver**. A **reply** suspends the context in which it is found and resumes the object which originally sent it a message. **Send** messages may be ordered in time or be indifferent to sequence. "Conditional branching" chooses one path to follow from many depending on a test of some kind. "Repeats" of various kinds cause evaluation to happen over and over; they may be terminated or restarted.

The independent state and message properties of Smalltalk make it possible to construct arbitrary structures or control structures.

Here are some of the abilities which have already been built for you to use. In the table below, the word *joe* is the name of an object that has been created. In creating a Smalltalk object, an entry is formed in a dictionary; each entry has two parts--the name of the object and the value of the object. Typically, the object has value as a class or as an instance of a class. As explained in previous chapters, class definitions have information known locally to the class as a whole (class variables) or to each instance of the class individually. Information known locally to each instance is either retained as part of the description of the instance (instance variables) or exists only when the instance is actively doing something (temporary variables). Dictionaries exist at each level of definition and activation of classes and their instances: there is a "global" dictionary known to all objects, one for each class, one for each instance of each class, and one for each object currently active.

Message Form	Meaning
joe b c	Send the object joe the message b c. Any message can be terminated with a period (.). There will always be a reply of some kind.
joe	Send the object joe an empty message. Usually the reply will be just a reference to joe's value.
☞*joe*	The "hand", ☞, says consider the next token literally--i.e., the literal word 'joe' instead of the object joe. A literal word is simply a string of characters; an object, however, refers to its value as a class or class instance. Here ☞ is an object being sent the message joe, and the reply is the literal word 'joe'.
☞*(a b ...)*	The reply is the literal chain (or vector) (a b ...).
◁*grow*	look (◁) in the message to see if the token (grow) is literally there. The reply will be 'not-false' if the token grow is literally there and the next thing in the message will now be available for scrutiny. Otherwise, the reply will be 'false' and whatever was there is still available.
:	The reply is the value of the next expression in the message.
⚷	The reply is the next literal token in the message.
⚷	Same as ⚷ except that the current place in the message will be retained regardless of the result of gathering the next token. This allows the receiver to "peek" at the message.
#	The reply is a reference to the meaning (class or class instance) of the next expression in the message. So, for example, if we have ☞func ← #hp, then the value of func is a reference to the meaning of hp; i.e., if hp is a class definition, then func becomes another name for the definition hp. Hence, mentioning func is identical to mentioning hp.

The user can construct other ways to receive messages from these primitives (such as "receivers" which check the class of the received object, and so on).

⇑ *3+4*	reply (⇑) to the sender the value of '3+4' which is 7; the 7 can now examine the current message.
a ⇒ (b) c ⇒ (d) ...	if a evaluates to 'not-false' then evaluate b and continue evaluation after the next enclosing parentheses. Otherwise evaluate c; if it replies 'not-false', evaluate d and continue evaluation after the next enclosing parentheses. Otherwise ...

The conditional expression *a ⇒ (b)* may be used anywhere in Smalltalk. Don't forget about the "escape" from the 'not-false' branch! If you would like to deliver one value or another depending on a condition, enclose the expression in '(...)'. Parentheses in Smalltalk serve a grouping or delimiting function: they delimit the 'then-clause' from the rest of a conditional expression; they delimit message parts to disambiguate or order the evaluation of the message; they group expressions for iteration using *repeat* or *do*; in general, they group a sequence of words together as a token that is received when the symbol ⍈ is used.

3+(a<b ⇒ (4) 5)

will evaluate to *7* or *8*, depending on the values associated with *a* and *b*. Here the outermost set of parentheses is used to order the evaluation of the message; the innermost parentheses define the limits of the 'then-cluase' for the conditional statement. Some examples of conditionally structured evaluation include:

evaluating a or b but not both	a ⇒ () b
letting evaluation of c depend on a or b	(a ⇒ () b) ⇒ c
letting evaluation of c depend on a and b	(a ⇒ (b)) ⇒ c

repeat (...)	The contents of () will be re-executed until a 'done' is encountered (or if you hit 'ESC'). The escape will be from the innermost loop in which the 'done' is enclosed.
done	Will cause the most recent repeat-loop to be exited.
done with 3+4	Will cause the most recent repeat-loop to be exited with the value 7 as a reply.
again	Will restart the most recent repeat-loop in which the again resides.
for	An iteration control feature included in the basic Smalltalk system. for i ← 2 to 50 by 4 do (...) Contents of ( ) will be re-executed until the value of index i, starting at 2 and stepped by 4 each time, exceeds 50. In general, the '←' part may be omitted and the default index start is 1; the 'by' part may be omitted and the default step is 1. If the 'to' part is omitted, the end condition value is the same as the start index value.
do n(...)	The contents of ( ) will be re-executed until the index counter N, starting at 1, equals n (i.e., for n ← 1 to n by 1). The counter N is not available as a number to use inside the parentheses.

Objects are created in one of two ways:

1. Creating a class

to ⟨class name⟩ ⟨temporary variables⟩ | ⟨instance variables⟩ | ⟨class variables⟩
(messages and responses)!

2. Creating an instance of a class

☞ *⟨name⟩ ← ⟨value⟩!*

where *⟨value⟩* is either the result of activating a class or activating an instance.

Other available (basic) abilities are described in subsequent sections.

The User Task

Smalltalk has a USER task which is evaluated continually. You can see the message that is evaluated by typing

t ← GET USER DO. !	Get the USER task.
t print.!	Ask to see the message.

In a Smalltalk system that does not include the dialog window class, the reply is

(cr. read eval print)	The reply is a vector, a request to evaluate a typed message.

The task shown above effectively:

(1) prints a carriage return in the Smalltalk dialog window (*cr*);
(2) prints the Interim Dynabook prompt character (Ω), reads characters from the keyboard until the <do it> character (!) is typed,
(3) assembles the characters into a list we call a vector;
(4) this vector is then an object that receives the message *eval print*; after seeing (*eval*) the token *eval*, it evaluates its contents as a message; and then
(5) whatever object the vector returns can receive the remaining message *print*. Some object is always returned, possibly the object *nil* (an object without value). The default object returned from running (activating) a class is the class instance (referred by the name SELF).

Some Comments. The routine *read* expects to print the characters typed at the keyboard in a dialog window whose name is *disp*. Vectors only respond correctly to the message *eval* when the last item in the vector is *nil*; hence the length of a vector containing Smalltalk message tokens ("code") is one item longer than the number of message tokens in the vector.

Effect of the Message Print and the Period. In order to fully understand the results of messages sent to Smalltalk, it helps to understand the implications of the *print* message. As an example, if you simply type a number or an arithmetic expression, without explicitly telling the resulting number to print itself, the number will, in fact, print. Try

3+4!	Reply is the number printed.

Now try

(3+4) print!	Reply is the number 7 printed twice without an intermediate space.
3+4.!	Note the period. Nothing seems to happen. The last message evaluated in the code vector is a period; the period returns itself as the reply; it then receives the message print and does nothing.
(3+4) print.!	The number 7 sees the message print and prints itself in the dialog window; the next token is a period; the period receives the print message (from the USER task); hence only one 7 prints.

This means that any object obtained as a result of evaluating a message at the top-level of Smalltalk will be sent the message *print* unless the original message is terminated with a period. If the resulting object does not respond to the message *print*, Smalltalk runs a "dummy" class named *print* which does nothing. Unexpected results might occur if the object does respond to the *print* message and the receipt of this message was not intended.

If you look at the USER task in a Smalltalk system with the dialog window class running, you will see the following (code) vector:

(sched map ☞(☞task ← vec[i]. apply task to ☞(run) in GLOB)!

This USER task assumes that there is an object named *sched* (an instance of the class *obset*), and that that object contains references to other objects (for example, dialog windows and/or font windows), each of which should receive the message *run* each time the USER task is evaluated. The usual response to the message *run* is to check to see if there is any keyboard input (*kbck*) and, if so, to evaluate the message *(cr. read eval print)*.

More information about this task is provided in the Chapter V section entitled *Scheduling Methods: sched and window.*

Active and Passive Return. We mentioned that the result of evaluating a message is a Smalltalk object that can receive the message *print*, unless a message terminator (a parenthesis or period) is used. Here we are saying that the result of evaluating a message is some value, an object that might be able to further examine the message.

This ability to let an object further examine the message depends on the method used to return it to the message sender. There are two methods for returning a value: passive return and an active return. The former is the default case--every evaluation results in some object whose value is, perhaps, *nil*. That object is returned to the message sender. Because it is returned passively, the object can not further examine the remaining message, if any.

The method of active return requires an explicit request to return the object. The Smalltalk symbol up arrow (⇑) is this explicit request. The form is ⇑ *⟨value⟩*; the *⟨value⟩* is an object that can examine the rest of the message. All numbers return actively; the class *turtle* returns its instance values actively (hence their ability to cascade messages). By default, instances return themselves passively unless the definition includes ⇑ *SELF* as a response to each message. The class *vector* receives the message *eval* and actively returns the result. Hence, the result of *read eval* is an object that can receive the next message: *print*.

The Form of Presentation of Smalltalk Classes In the next chapter, we present definitions for the basic Smalltalk system: the classes already defined for general use, aids for interacting with Smalltalk and with the Smalltalk file system. Chapter V contains examples of applications of these classes. The basic Smalltalk classes will be presented by showing how instances of each class are created and what happens when messages are sent to a class instance. In most cases, the messages are annotated; in some cases, the actual definition of the class will be shown. For example, a version of the class *box* defined in Chapter II can be presented as:

box The name of the class.

☞*joe ← box*! Creating an instance of the class.
I'm a box : x 256 y 300 size 50 tilt 0

joe is ?! What is the instance type.
box

joe is box! 'Not-false' is the same as 'true'.
box

joe's x ← 200! Assigning meaning in joe's context.
200

joe's x! Querying joe's context.
200

joe's y ← 250!
250

joe's y!
250

joe's size ← 100!
100

joe's size!
100

joe's tilt ← 32!
32

joe's tilt!
32

joe draw!

joe undraw!

joe grow 3+4!

*joe turn 20*2*!

joe move 100 200!

Abbreviations

In order to present these examples a bit more concisely, we need to adopt some abbreviations.

We Abbreviate	By
a property of a class instance	⟨property⟩
expected value (any type)	⟨value⟩
expected number value	⟨number⟩
expected nonnegative integer value	⟨integer⟩
instance of a class named classname	⟨classname⟩
name of an object	⟨name⟩
expected string value	⟨text⟩
expected message stream	⟨message⟩
forms involving [ ]	⟨selection⟩

We can further simplify the presentation of classes if some class conventions are adopted, such as: all classes will respond reasonably to the following messages:

is ?	replies with *⟨classname⟩*
is ⟨classname⟩	replies *⟨classname⟩* or *false*
print	prints in standard format
's ⟨property⟩ ← ⟨value⟩	makes *⟨property⟩* stand for the *⟨value⟩*
's ⟨property⟩	replies with *⟨value⟩* of *⟨property⟩*

Class *box* then can be described compactly as:

box

☞*joe ← box*!	Draws a square at x = 256, y = 300, size of each side = 50, and angle of tilt = 0.
joe draw!	
joe undraw!	
joe grow ⟨number⟩!	joe erases, makes himself bigger by ⟨number⟩ units, and redraws.
joe turn ⟨number⟩!	joe erases, turns himself by ⟨number⟩ degrees and redraws.
joe move ⟨number⟩ ⟨number⟩!	joe erases, changes his coordinates, and redraws in a new location.

A Smalltalk Class Example

Link is a typically complete form which we present as an example of the conventions for presenting a class definition. It is a structure familiar to LISP users: pairs of objects which may in turn also be pairs. Instances of *link* receive and respond to the following messages.

link init!	Set up help and mail box information.
☞*pair* ← *link* ☞*john* ☞*mary*.!	Create an instance whose name is pair (or, as in LISP, "cons").
pair head! *john*	Ask for the value of instance variable h (or, as in LISP, the "car").
pair tail! *mary*	Ask for the value of instance variable t (or, as in LISP, the "cdr").
☞*triangle* ← *pair* + ☞*jim*.!	Create another instance whose head is the instance pair and whose tail is ☞jim.
triangle print.! *((john . mary) . jim)*	Show the value of triangle.
triangle is ?! *link*	triangle is an instance of what class?
pair lprt! *I am a link.* *I consist of (john . mary)*	Provide some helpful information about the instance pair.

The form of the class definition is

```
to link a | h t | helpprint mailbox

   ( ◀+ ⇒ (⇑ link SELF : )

     ◀head ⇒ (⇑ (◀← ⇒ (☞h←:.) h))

     ◀tail ⇒ (⇑ (◀← ⇒ ( ☞t←:.) t))

     ◀lprt ⇒ (helpprint SELF)

     ◀print ⇒ (disp←'('. h print. disp←'.'. t print. disp←')'.)

     ◀is ⇒ (◀ link ⇒ (⇑ ☞link) ◀ ? ⇒ (⇑ ☞link) &. ⇑false.)

     ◀'s ⇒ (☞a ← &. ⇑ (◀ ← ⇒ (a ← :) a eval))

     ◀init ⇒ (☞helpprint ← #hp. ☞mailbox ← 'no mail'.)

     isnew ⇒ (☞h ← :. ☞t ← :.)!

  to hp ob

    (☞ob ← :. cr. disp ← 'I am a ' . (ob is ?) print.

      cr. disp ← 'I consist of '. ob print.)!
```

Chapter IV. BASIC SMALLTALK SYSTEM CLASSES AND UTILITIES

The Basic System Classes

See the end of Chapter III for an explanation of the method for presenting the basic Smalltalk system class definitions.

Atoms

Smalltalk atoms are unique tokens which are usually associated with Smalltalk objects in dictionary entries. If a user attempts to create an atom which will print the same as an already created atom, the system will force the two to be the same.

atom

☞*a* ← ☞*b*! *b*	The value of a is the atom b.
☞*a* ← *atom* ‹*text* ›!	Reply is the new name which prints as ‹text›.
a chars! *'b'*	Reply is the ‹text› of names value.
a ← ‹*value*›!	The ‹value› is associated with the name b (i.e., this is indirect reference to the name b).
a! *b*	The value of a is b.
b! ‹*value*›	The value of b is ‹value›.
a eval! ‹*value*›	Indirect reference--a eval is the value of a which is b, and the value of b prints, which is ‹value›.
a = ‹*name*›!	Value of a if 'not-false', 'false' otherwise.

Arithmetic

There are two classes for handling numerical operations: *number* and *float*. They are compatible and interchangable. An operation containing both classes will have a reply in the class of the first object (that is, in the class of the object being sent the message).

100/8.0!
12

100.0/8!
12.5

The value range of *number* is

-32768 to 32767

that of *float* is (where the form 1.2e3 denotes 1.2 times (10 to the power 3))

-99999.99999e4095 to 99999.99999e4095

An *integer* beginning with the digit 0 is an octal number; all other numbers are base 10. *float* must begin with a digit from {0, ..., 9}. *float* must have an embedded period, *numbers* must not. In addition, *float* may be expressed in scientific notation as a product of a power of 10.

Good Forms	Bad Forms
123	
-123	
0.0	*.0*
355.0	*355.*
6.28e-23	*28e-23*

number

a ← 128! *128*	Value of a is 128, a number.
a + (number)!	Reply is the numeric sum of the two objects.
a - (number)!	Reply is the numeric difference of the two objects.
*a * (number)*!	Reply is the numeric product of the two objects.
a / (number)!	Reply is the integer quotient of the two objects.
a mod (number)!	Reply is the integer remainder.
- a!	Reply is the numeric negative of a. The unary minus is typed holding down the <shift> key and pressing -.
a = (number)!	Reply is the value of a if 'not-false', otherwise 'false'.
a ≠ (number)!	Reply is the value of a if 'not-false', otherwise 'false'.

a < *⟨number⟩*!	Reply is a if 'not-false', 'false' otherwise.
a ≤ *⟨number⟩*!	Reply is a if 'not-false', 'false' otherwise.
a > *⟨number⟩*!	Reply is a if 'not-false', 'false' otherwise.
a ≥ *⟨number⟩*!	Reply is a if 'not-false', 'false' otherwise.
a ☐ *⟨number⟩*!	Reply is the bitwise logical operation of the two values.
a☐*⟨number⟩*!	logical AND
a☐*⟨number⟩*!	logical OR
a☐*⟨number⟩*!	logical XOR
a☐*⟨number⟩*!	LSHIFT by the ⟨number⟩
a min⟨number⟩!	Reply is the minimum of the two values.
a max ⟨number⟩!	Reply is the maximum of the two values.

In the above, *⟨number⟩* can be an instance of *number* or of *float*, but the result is the proper *number* result.

float

☞*a* ← *3.14159*!
3.14159

☞*a* ← *float ⟨number⟩*!	Reply is the floating point equivalent of the number.
a + *⟨number⟩*!	In the following, reply is the proper floating point result, but ⟨number⟩ can be an instance of number or of float.
a - *⟨number⟩*!	
a * *⟨number⟩*!	
a / *⟨number⟩*!	
- *a*!	
a = *⟨number⟩*!	
a ≠ *⟨number⟩*!	
a < *⟨number⟩*!	
a ≤ *⟨number⟩*!	
a > *⟨number⟩*!	
a ≥ *⟨number⟩*!	
a ipart!	Reply is the integer part of the floating point number; can not be in scientific notation. E.g., 27.3 ipart! Reply is 27.

a fpart!

Reply is the fractional part of the floating point number; can not be in scientific notation.
E.g., 27.3 fpart! Reply is 3.

a ipow <number>!

Reply is the result of a to the power <number>.

a epart <float>!

Reply is X where X ipow <float> = a .
E.g., 27.0 epart 3.0!
Reply is 3.0.
This is used for printing floating point numbers.

Turtles for Drawing

A *turtle* is a method for drawing on the display screen. The class *turtle* was introduced earlier in Chapters I and II. *Turtles*, like 😊, can receive any number of cascaded messages. For example,

😊 *penup goto 200 300 pendn*!

is equivalent to:

😊 *penup.*
😊 *goto 200 300.*
😊 *pendn*!

However, there is no cascading after the 's message. A turtle's width can vary from 0 to 8 dots. Say:

😊's *width* ← *4.* 😊 *go 100*!

turtle

👉😊 ← *turtle frame ⟨dispframe⟩*! 👉😊 ← *turtle*!	Turtle's range is defined by the boundaries of the dispframe. Turtle's range is the entire display screen.
😊 *home*!	Picks up pen, takes 😊 to geometric center of range, faces upward.
😊 *erase*!	Erases range.
😊 *up*!	Faces turtle towards top of display screen.
😊 *penup*!	Any travelling will not leave a trace.
😊 *pendn*!	Any travelling will leave a trace if ink is different from background.
😊 *black*!	Sets ink to black.
😊 *white*!	Sets ink to white.
😊 *xor*!	Trail exclusive-or-ed with other stuff on screen, if width=1.
😊 *go ⟨number⟩*!	Travels in current direction a distance ⟨number⟩.
😊 *turn ⟨number⟩*!	Turns clockwise ⟨number⟩ degrees from current direction.
😊 *goto ⟨number⟩ ⟨number⟩*!	Travels to x = ⟨number⟩, y = ⟨number⟩.
😊 *goto ⟨point⟩*!	Travels to the place represented by the point and does not change its direction.
😊 ← *⟨text⟩*! 😊 ← *⟨integer⟩*!	Prints the text (or the character represented by the Ascii code ⟨integer⟩) at the turtle's current location, with its direction, width and color.

The False Class

is a method for handling boolean operations.

false

bool ← *false*!

bool ⇒ (*⟨message⟩*)!	Since bool is 'false', gathers up the message without evaluating.
bool or ⟨message⟩!	Reply is result of evaluating ⟨message⟩.
bool and ⟨message⟩!	Evaluates message; reply is SELF.
bool < *⟨message⟩*!	Evaluates message; reply is SELF.
bool = *⟨message⟩*!	Evaluates message; reply is SELF.
bool > *⟨message⟩*!	Evaluates message; reply is SELF.

Sequential Dictionaries

include the classes: *vector, string, obset, stream, file.*

Vectors and Strings

are both organized like beads on a string. Their only difference is the way they respond to:

is ?

and that a *vector* may have any Smalltalk object as a bead while *string* may only contain whole numbers ranging from 0 to 255. *String* objects are thus not absolutely necessary (since *vector* beads can contain any Smalltalk number), but are very useful as a compact way to store textual information. The characters you type to Smalltalk are first captured as a *string* object and the textual information which Smalltalk shows you is held as a *string* object belonging to a *dispframe* object. To save space, the messages of both these classes will be shown together, repeating messages in the separate columns only when expected values and replies differ.

vector	*string*
☞a ← ☞(this is a vector literal)!	*☞a ← 'this is a string literal'!*
(this is a vector literal)	*'this is a string literal'*
☞a ← vector ⟨number⟩!	*☞a ← string ⟨number⟩!*
	Objects of the class are created with initial length ⟨number⟩.
a[⟨number⟩]!	Reply is the value of the bead found at position ⟨number⟩ Note that the first position is 1, not 0.
a[⟨number:lb⟩ to ⟨number:ub⟩]!	Reply is a 'subvector' or 'substring' of beads whose values are copied starting at ⟨number:lb⟩ (lower bound) and ending with the value at ⟨number:ub⟩ (upper bound). We call either of the forms involving *[ ]*, [⟨number⟩] and [⟨number:lb⟩ to ⟨number:ub⟩], a ⟨selection⟩.
a ⟨selection⟩ ← ⟨value⟩!	If the ⟨selection⟩ is of a single element, the value of the bead found at position ⟨number⟩ becomes ⟨value⟩. Otherwise, ⟨value⟩ is expected to be a string of beads of the same class as a and of any length. The ⟨selection⟩ is replaced by the ⟨value⟩.
a ⟨selection⟩ ← ⟨value⟩ ⟨selection⟩!	The form ⟨value⟩⟨selection⟩ is a method for obtaining a string of beads of the same class as a
a ⟨selection⟩ ← all ⟨value⟩!	Copies the ⟨value⟩ into each element in the selection. This was used in the sketching example in Chapter II: cobweb.
a ⟨selection⟩ find first ⟨value⟩!	Reply is the first bead position ⟨number⟩ where a[⟨number⟩] is the same as ⟨value⟩ if a[⟨number⟩] is found in the range of the ⟨selection⟩, 0 otherwise.
a ⟨selection⟩ find first non ⟨value⟩!	Similar to previous, except elements of ⟨value⟩ are ignored.

a ⟨selection⟩ find last ⟨value⟩!	Reply is the last bead position ⟨number⟩ where a[⟨number⟩] is the same as ⟨value⟩ if a[⟨number⟩] is found in the range of the ⟨selection⟩, 0 otherwise.
a ⟨selection⟩ find last non ⟨value⟩!	Similar to previous, except elements of ⟨value⟩ are ignored.
a eval!	Vectors only. Treats the contents as Smalltalk code. Evaluation is in current context; last item of vector must be nil.
a length!	Reply is the number of bead positions
a + ⟨vector⟩!	Joins copies of a and ⟨vector⟩ (⟨string⟩) into a new vector
a + ⟨string⟩!	(string).
a map ⟨vector⟩!	Vectors only. The value of ⟨vector⟩ is sent as a message to each of the beads of a.
a = ⟨string⟩!	Strings only. Reply is ⟨string⟩ if a is identical to ⟨string⟩; otherwise false.

Obsets

Obsets are "bushel baskets" which can hold things for you. They can be used like mathematical sets (having only unique values) or like "bags" (being able to contain duplicate values). Instances of *obset* are frequently used as schedulers for the objects which they contain. For example, the display windows of various kinds are all contained in an *obset* called *sched.* An instance of *obset* owns its own instance of *vector* and provides a method for automatically expanding the *vector*, storing objects in the next available position in the *vector*, and removing objects.

obset

☞ob ← obset!	An instance of obset is given the name ob.
ob ← ⟨value⟩!	If the ⟨value⟩ is not already in ob it will be added, otherwise ob stays the same. This addition method (set union) depends on checking for equivalence of the values in ob. Since ob actually contains pointers to the Smalltalk objects, large integers of the same value will typically not be equivalent, as their pointers are not equivalent.
ob delete ⟨value⟩!	Assuming there is only one occurrence of ⟨value⟩, it will be deleted if in ob; if there are multiple occurrences, only the first will be deleted; reply is 'false' if there is no occurrence.
ob add ⟨value⟩!	The ⟨value⟩ is added whether or not one already exists there.
ob unadd!	The most recently added ⟨value⟩ will be deleted. add and unadd can be used to implement a "stack".
ob vec!	Reply is a vector containing all the objects of ob.
ob map ⟨vector⟩	Evaluates the ⟨vector⟩ n times where n = the number of objects in ob's vector.

An *obset* is one method of using vectors. Objects in an *obset* are actually stored in a vector that is locally bound to the instance of the *obset*. The vector instance is named *vec*; *i* is the index counter used in replying to the message *map*. Hence, if we wanted to send every object in the *obset sched* the message *run*, we would say

sched map ☞(vec [i] run)!

where vec[i] refers to the ith object in the obset. It is also possible to refer to each object by the object *each* so that the above message could be written as

sched map ☞(each run)!

Many users add their own version of intersection, union, and so on, to the definition of *obset*.

Streams

Streams are similar to the BCPL programming language method for storing and retrieving information. A pointer, *i*, is kept to the current *stream* item; pointer *L* keeps track of the last storable item. The actual storage method is either a *string* or a *vector* bound to the instance of *stream*. We use double quotes " to indicate optional forms.

☞*s* ← *stream*!	Default is to create storage in a string of length 10; i=0; L=10.
☞*s* ← *stream of vector* ⟨*m*⟩!	Create storage in a vector of length m; i=0; L=m.
☞*s* ← *stream of string* ⟨*m*⟩!	Create storage in a string of length m; i=0; L=m.

☞*s* ← *stream "of vector* ⟨*m*⟩*" "from* ⟨*integer1*⟩*" "to* ⟨*integer2*⟩*"*!
☞*s* ← *stream "of string* ⟨*m*⟩*" "from* ⟨*integer1*⟩*" "to* ⟨*integer2*⟩*"*!

Initially, s is either a string or vector referenced starting before the first item (i=0) up to the last storable position (L= length of the string or vector). Or, optionally, s may be a different length string or vector (m) whose contents are referenced beginning with an index other than 0 (i= ⟨integer1⟩ - 1) up to an index other than the actual string or vector length (L = ⟨integer2⟩).

s ← ⟨*value*⟩!	Stores in the next (☞i←i+1) item of the stream, expanding the length of the stream if i=L.
s contents!	Returns the stored items (from the first up to the ith item).
s next!	Returns 0 if i=L; otherwise, returns the i+1st item and increments i.
s reset!	Resets i to 0 (points to the beginning of the stream)
s end!	Returns 'true' if i is the end of the stream (i=L); otherwise returns 'false'

Files

The Smalltalk file system provides for instances of the class *directory* divided into files. A *file* is found in a directory by its file name (*fname*). A file name must be an instance of the class *string*. Each file has in its local context a character pointer (*bytec*) and a 512-character string as an i/o buffer (*sadr*). Each file also knows the directory in which it can be found (*dirinst*).

Initially, there are two directories: *dp0, dp1*. However, only *dp0* should be used unless the Interim Dynabook is equipped, for example, with two Diablo model-31 disk drives or with a Diablo model-44 disk. When creating a file instance, you actually send a message to an instance of the class *directory*. Effectively, this sets the instance of the *directory* as the value of *curdir*. If the directory reference is omitted, Smalltalk runs the class *file* with *curdir* equal to *nil*, indicating that the directory should be the default name stored as *defdir*. Unless specified, *defdir* is defaulted to *dp0*. To modify this, type

⟨directory⟩ use!

In the following, "<directory>" is therefore optional.

Creating File Instances

☞fi ← ⟨directory⟩ file ⟨text⟩ old!	Searches for a file previously defined in the directory; returns 'false' if not found.
☞fi ← ⟨directory⟩ file ⟨text⟩ new!	Creates a new file or returns 'false' if a file with the same name already exists.
☞fi ← ⟨directory⟩ file ⟨text⟩!	First attempts to find an old file; if it fails, then creates a new file.
⟨directory⟩ file ⟨text⟩ exist!	Answers the question, does the file already exist in the directory?

Deleting a File

⟨directory⟩ file ⟨text⟩ delete!	Deletes the file if it exists; returns 'false' otherwise.

Renaming a File

⟨directory⟩ file ⟨text⟩ rename ⟨text⟩!

Loading and Saving Entire Smalltalk Context

⟨directory⟩ file ⟨text⟩ load!

⟨directory⟩ file ⟨text⟩ save!

Interrogating the Directory

⟨directory⟩ list!	Will print the names of all the files on the directory.

Reading and Writing a File

It is possible to read and write strings, words, or characters from a file. A word is simply two characters on even character boundaries, while a string is a set of n characters. In the following, local context for a file instance includes:

leader	disk address of page 0
curadr	disk address of current page
nextp	disk address of next page
sadr	512 character string
bytec	character index into sadr
numch	number of characters on the current page, must be 512 unless current page is the last page
pagen	current page number
sn1,sn2	unique 2 word serial number for the file
version	version number, currently always 1

fi ← ⟨integer⟩!	Store a number (Ascii code).
fi ← ⟨text⟩!	Store each character in the string onto fi.
fi next!	Read the next character from fi (8 bits).
fi next word!	Read an integer from fi (16 bits). Adjusts character pointer to retrieve the logical next word.
fi next word ← ⟨number⟩!	Write the number into the next word of fi.
fi next into ⟨text⟩!	Read enough characters from fi to fill the string ⟨text⟩. This is essentially, but not identical code as, for j to ⟨text⟩ length do (⟨text⟩[j] ← fi next)
fi flush!	Write out sadr (the i/o buffer) onto fi.

Addressing a File

fi skipnext ⟨number⟩!	Relatively positions a file. Same as fi set to read pagen bytec + :.
fi end!	Returns file instance if pagen, bytec points to the end of the file; returns 'false' otherwise.
fi shorten to ⟨integer⟩ ⟨number⟩!	Set nextp to 0, pagen to integer, bytec and numch to number.
fi shorten to here!	Same as fi shorten to current file location, i.e., pagen bytec.
fi print!	Prints the file name.
fi reset!	Same as fi set 1 0 (point to beginning of file).
fi set to write ⟨integer⟩⟨number⟩!	Sets bytec to number; pagen to integer; allocates new pages if try to go beyond the end of file.
fi set to read ⟨integer⟩ ⟨number⟩!	Same as write but will stop if try to go beyond the end without allocating new pages.
fi set ⟨integer⟩⟨number⟩!	Same as set to read.

fi set to end! Same as fi set to read 037777 0 (i.e., forces end of file).

Files Open List

A list of file and directory instances currently being referenced for each directory is kept in a "files open list".

⟨directory⟩ print! Prints the entry names of each open file in the directory.

⟨directory⟩ flush! Write out the current state of each file in the filesopen list.

⟨directory⟩ close! Flush the directory and reset the filesopen list.

Individual files can be added or removed from the files open list.

⟨directory⟩ remember ⟨value⟩!

⟨directory⟩ forget ⟨value⟩!

fi remove! Remove file from the files open list.

fi close! Remove file from the files open list and flush the bittable and the current page.

Dispframe: The Basic Window Class

Text Display Routines

Smalltalk has a multiple-window display capability which allows viewports composed of text, pictures, musical notation, and so on, to be created. The main method for creating and editing windows of text is to create instances of the class *dispframe*. These display frames are rectangular areas on the screen. They are specified with five values: an upper left corner horizontal position x, a width, an upper left corner vertical position y, and a height. A fifth value specifies either an instance of class *string* or creates the instance by including the words *string ⟨integer⟩*. Hence

```
df ← dispframe 16 256 16 256 string 400!
```

gets you a rectangular area on the upper left portion of the display screen. The upper left corner is 16,16; the width and height are 256; and a string of 400 characters (whose local name is *buf*) serves as the text buffer. This buffer is altered by ← (store characters) and by scrolling in the window. Or,

```
ef ← dispframe 3 100 50 200 ' ' !
```

gets you a rectangular area at upper left corner 3,50 with a width of 100 and height of 200. The buffer is a string with length 1. The instance variable *last* is set to 0. It is possible to create a *dispframe* by stating the actual text of the frame, i.e.,

```
gf ← dispframe 3 100 50 200 'hello there'!
```

However, the text will not show because the index into the text string is *last* = 0, indicating that no characters are to be shown.

There are actually two entities associated with a display frame: a frame and a window. Clipping and scrolling are done on the basis of window boundaries. Window boundaries are intersected with the physical display screen. The frame may be smaller or larger than the window and smaller or larger than the physical display screen. Frame boundaries are the basis for word-wraparound.

Presently, dimensions defining frame and window boundaries are given the same values upon creating an instance of *dispframe*. The following are local bindings (instance variables) for each instance of the class.

winx	window upper left corner x
winwd	window width
winy	window upper left corner y
winht	window height (note, automatically increased on creation of the instance to make the window extend to the bottom of the display screen)
frmx	frame upper left corner x
frmwd	frame width
frmy	frame upper left corner y
frmht	frame height
buf	string buffer
last	pointer to the current last character stored in buf
lstln	pointer to the character in buf that begins the last line of text in the frame
mark	pointer to the character in buf representing the last prompt output
charx	right x position of the character pointed to by index last
chary	top y position of the character pointed to by index last

reply	indicator for frame and window control (see below)
justify	toggle for right justifying the contents of the window 0 means no justification; 1 means justify on frame boundaries
font	font for displaying characters if nil, then default font used; otherwise, the value of font is a string defining the font to be used (see below)
editor	available storage for associating a unique editor with any display frame.

The text buffer *buf* contains only characters that can be displayed within the window boundaries. Scrolling occurs when an attempt to store more characters causes overflow of the bottom of the window. In this case, the first line of characters (where a line is defined according to frame boundaries) is stripped out of *buf*.

The *reply* variable is useful in controlling window and frame boundaries and scrolling. The following are meaningful values for *reply*:

0	everything is okay--there was intersection between window and display and between the window and the frame.
1	no intersection between window and display
2	no intersection between window and frame
3	window height less than font height so not even one text line can be displayed
4	frame height has been increased to accommodate new text
5	overflowed bottom of window (scrolling)
6	both 4 and 5 occurred

To get a different font other than the default font, it is necessary to read the font string from a previously created file (see section on Editfont on how to create fonts). Type

☞*ff ← file ⟨text⟩ intostring*!

Then, assuming the name of the dispframe is *disp*, say

disp's (☞*font ← ff*)!

Or, you can declare the font at the same time you create the instance of the *dispframe*.

☞*df ← dispframe ⟨integer⟩ ⟨integer⟩ ⟨integer⟩ ⟨integer⟩ ⟨text⟩*!
☞*df ← dispframe ⟨integer⟩ ⟨integer⟩ ⟨integer⟩ ⟨integer⟩ string ⟨integer⟩*!

Create an instance of dispframe with values for window and frame boundaries and length of the text buffer. The window will appear on the display screen with a black double line around it. In the first case, where a text string has been specified, it will not appear because the variable last is set to 0. It would be necessary to type

df's (☞last ← buf length). df display.!

to actually see the text.

☞*df ← dispframe ⟨integer⟩ ⟨integer⟩ ⟨integer⟩ ⟨integer⟩ ⟨text⟩ font ⟨fontstring⟩*!
☞*df ← dispframe ⟨integer⟩ ⟨integer⟩ ⟨integer⟩ ⟨integer⟩ string ⟨integer⟩ font ⟨fontstring⟩*!

Create an instance of dispframe with value for font.

☞*df ← dispframe ⟨integer⟩ ⟨integer⟩ ⟨integer⟩ ⟨integer⟩ ⟨text⟩ noframe*!
☞*df ← dispframe ⟨integer⟩ ⟨integer⟩ ⟨integer⟩ ⟨integer⟩ string ⟨integer⟩ noframe*!

Create an instance of dispframe with values for window and frame boundaries and length of the text buffer. Window will not have a black line around it.

df ← ⟨text⟩!
Append the string ⟨text⟩ to buf and display if possible

df ← ⟨integer⟩!
Append this Ascii character to buf and display its corresponding character if possible.

df show!
Clears the intersection of window and frame and displays buf.

df display!
Does a show, then draws double black line around the window.

df frame!
Draws a double black line around the window.

df frame black!
Same as df frame.

df frame white!
Draws a double white line around the window.

df frame color ⟨integer⟩!
(color display only) Draws double line around the window in color denoted by the integer number.

df hasmouse!
Returns 'not-false' if the mouse cursor is within the frame; otherwise returns 'false'.

df fclear!
Clears the intersection of the window and frame.

df wclear!
Clears the intersection of the window and the physical display.

df clear!
Does an fclear and then sets last to 0 and lstln to 1, effectively cleaning out the text buffer.

df scroll!
Removes the top line of text from buf and moves the text up one line in the frame.

df mfindc ⟨integer⟩⟨integer⟩!
Find character located at ⟨integer⟩,⟨integer⟩.
Returns vector vec such that

vec[1]	subscript of character in string
vec[2]	left x of character
vec[3]	width of character in string
vec[4]	top y of character

If vec[1]=-1 then position is after the end of string.
If vec[1]=-2 then position is not in the window.

df mfindw ⟨integer⟩ ⟨integer⟩!
Find word located at ⟨integer⟩,⟨integer⟩.
Returns vector vec such that

vec[1]	subscript of first character in word
vec[2]	left x of word
vec[3]	width of word
vec[4]	top y of word

If vec[1]=-1 then position is after end of string.
If vec[2]=-2 then position is not in the window.

df mfindt ⟨integer⟩ ⟨integer⟩!
Find token located at ⟨integer⟩,⟨integer⟩.
Returns vector vec such that

vec[1]	token count where spaces and carriage returns are considered delimiters but multiple delimiters do not increment the count. ⟨text⟩ counts as one token.
vec[2]	left x of token
vec[3]	width of token
vec[4]	top y of token

If vec[1]=-1 then position after end of string or not in frame.
If vec[1]=-2 then position is not in the window.

df read!	Makes a vector out of keyboard input. Assumes the name of the dispframe is disp.
df reread ⟨integer⟩!	(Used by fix and redo). Counts back from end of buf an ⟨integer⟩ number of prompts in the buffer and does a read from there.
df sub ⟨value⟩!	Evaluates ⟨value⟩ in the context of the dispframe. (Used by fix to evaluate the editor within the window and by shift-esc to create a window within the window).
df hide!	Same as df fclear. df frame white.
df put ⟨string⟩ at ⟨x⟩ ⟨y⟩!	Prints the text ⟨string⟩ starting at position x,y. Upper left corner of df becomes x,y.
df corner ⟨x⟩ ⟨y⟩!	Returns 0 if position x,y in no corner returns 1 if position x,y in upper left corner returns 2 if position x,y in upper right corner returns 3 if position x,y in lower left corner returns 4 if position x,y in lower right corner
df moveto ⟨x⟩ ⟨y⟩!	Set winx and frmx to ⟨x⟩; set winy and frmy to ⟨y⟩.
df growto ⟨x⟩ ⟨y⟩!	Set winwd and frmwd to (⟨x⟩ - frmx); set winht and frmht to (⟨y⟩-frmy).

The last three messages are added to *dispframe* when the window framework is included in the basic Smalltalk system.

Four routines are available for manipulating rectangular areas of the display.

dclear ⟨integer⟩ ⟨integer⟩ ⟨integer⟩ ⟨integer⟩ ⟨number⟩!

will clear the rectangular area defined by the four integers, where the order specifies:⟨upper left corner x⟩ ⟨width⟩ ⟨upper left corner y⟩ ⟨height⟩. The cleared area is then filled with black and white dots according to the binary representation of the number given (1's = black, 0's = white). For example, if the number is -1, the area will be all black.

dcomp ⟨integer⟩ ⟨integer⟩ ⟨integer⟩ ⟨integer⟩!

will complement the rectangular area defined by the four integers, where the order specifies: ⟨upper left corner x⟩ ⟨width⟩ ⟨upper left corner y⟩ ⟨height⟩.

dmove ⟨integer⟩ ⟨integer⟩ ⟨integer⟩ ⟨integer⟩ ⟨integer⟩ ⟨integer⟩ ⟨integer⟩!

will take the source rectangular area defined by the first four integers (same order as above), and move it to the destination defined by the fifth and sixth integers (destination upper left corner x,y). The seventh integer is a mode indicator: if the mode is 0, the source rectangular area will be stored as given; if the mode is not 0, the black and white dots in the source rectangle will be 'or-ed' with the dots in the destination area (0 or 0 = 0; 0 or 1 = 1; 1 or 0 = 1; 1 or 1 = 1).

dmovec ⟨integer⟩ ⟨integer⟩ ⟨integer⟩ ⟨integer⟩ ⟨integer⟩ ⟨integer⟩ ⟨integer⟩!

same as dmove except that the non-intersecting source rectangular area is cleared.

Point Class

A *point* is an example of a storage method. Several examples of its use have already been given in the Chapter II section on sketching ideas.

point

pt ← point 100 200!	Create a point whose horizontal coordinate is 100 and vertical coordinate is 200.
pt x! *100*	
pt y! *200*	
pt + ⟨point⟩!	Reply is point obtained by adding coordinates of pt and ⟨point⟩.
pt - ⟨point⟩!	Reply is point obtained by subtracting coordinates of pt and ⟨point⟩.
pt = ⟨point⟩!	Reply is pt y if they are the same points, otherwise false.
pt ≤ ⟨point⟩!	Reply is pt y if pt is a point whose horizontal and vertical positions are smaller or equal to those of ⟨point⟩.
pt max ⟨point⟩!	Reply is a point whose horizontal position is the maximum of that for pt and ⟨point⟩; similarly for the vertical position.
pt min ⟨point⟩!	Reply is a point whose horizontal position is the minimum of that for pt and ⟨point⟩; similarly for the vertical position.

This class is provided partly at the machine code level. The corresponding code is equivalent to

```
to point a | x y
    (isnew ⇒ (☞x ← :. ☞ y ← :.)
      ✍ x ⇒ (✍← ⇒ (☞x←:.) ⇑ x)
      ✍ y ⇒ (✍ ← ⇒ (☞ y←:.) ⇑ y)
      ✍+ ⇒ (☞a ←:. ⇑ point x+a x y+a y)
      ✍ - ⇒ (☞a←:. ⇑ point x - a x y - a y)
      ✍ = ⇒ ((☞a←:) ⇒ (⇑ false) x = a x ⇒ (⇑y = a y) ⇑ false)
      ✍ ≤ ⇒ ((☞a←:) ⇒ (⇑ false) x  a x ⇒ (⇑ y  a y) ⇑ false)
      ✍ max ⇒ (☞a←:. ⇑ point (x max a x)  (y max a y))
      ✍ min ⇒ (☞a←:. ⇑ point (x min a x) (y  min a y))
      ✍ print ⇒ (☞point print. sp. x print. sp. y print))!
```

Also provided in the basic Smalltalk system is the routine *mp*

```
to mp  (⇑ point mx my)!
```

Aids for Interacting with Smalltalk

The Smalltalk Class Editor

edit ‹classname›!

will get you the Smalltalk editor for the class which is named ‹classname›.

fix ‹integer›!

where integer is the number of transactions (images of the Interim Dynabook) back from where you are, will get you the Smalltalk editor for transaction integer. Upon exiting, the edited transaction will be evaluated, but the original transaction will not be modified.

edit ‹classname› title!

will start the editing with the title line.

The editor shows two frames. The righthand frame contains a menu of commands, the left hand frame contains a structured representation of the definition. All tokens at a single level of parenthezation are shown. A lower level of parentheses is shown as (). An example is:

do 4 (☺ go 100 turn 90)

is shown as

do 4 ()

All editing is done by "grabbing" a command in the righthand menu (pointing to it with the cursor and pushing the top or middle mouse button).

In the following, "text" refers to characters typed from the keyboard and terminated with !.

Commands	Number of Times Grabbing Needed	Action Taken
Add	0	Append text to end of current level.
Insert	1	Add text before designated word.
Replace	2	Replace the text indicated by pointing to the beginning and end words with new text.
Delete	2	Delete the text indicated by pointing to the beginning and end words.
Move	3	Combination of deleting text and inserting new text before the word pointed to as third 'grab'.
Up	1	Remove parentheses.
Push	2	Put parentheses around words indicated by pointing to the beginning and end of the intended grouping.
Enter	1	See the next lower level.
Leave	0	See the next higher level.
Exit	0	Terminate editing.

The only exceptions are Enter and Up. If there is only one level marker, (), showing in the current level, no grabbing is required.

Showing Stored Information

show ⟨name⟩!
will show you what meaning the ⟨name⟩ currently has.

defs!
will show you the names of classes you have defined that are currently available.

dp0 list!
will show you the names of files stored on your disk pack.

type ⟨text⟩!
will show you the contents of file named ⟨text⟩; returns 'false' if the file does not exist.

Saving Smalltalk Definitions

filin ⟨text⟩!

will go to a file whose name is ⟨text⟩ and tell Smalltalk to read what it finds on the file. Example:

filin 'boxes'!

Usually the file will contain programs written there by running *filout* as defined next.

filout ⟨text⟩!

will write every program whose name is in *defs* to a file called ⟨text⟩. Example:

filout 'boxes'!

will write out every program whose name is currently in *defs*. There are a few other useful variations of *filout*.

filout ⟨text⟩ ⟨vector⟩!

will ignore *defs* and only write out the programs mentioned in the *vector*. Example:

filout 'boxes' ☞*(box square triangle)*!

will ignore *defs* and only write out the three programs whose names appear in the vector.

Suppose the vector contains vectors, for example,

filout 'boxes.'
☞*(boxes square (addto turtle* ☞*(*◀*place ⇒ (SELF penup goto (:)(:) pendn up))))*!

will write out the programs *boxes* and *square*, and then the vector

(addto turtle (◀*place ⇒ (SELF penup goto (:)(:) pendn up))).*

On filing in this file, the ability to receive the message *place* will be added to the class *turtle.*

filout pretty ⟨text⟩!
filout pretty ⟨text⟩ ⟨vector⟩!

will format the programs so they will print nicely (in *show* format).

filout ⟨text⟩ add!

will not overwrite file ⟨text⟩ but instead will add the new definitions at the end of ⟨text⟩. Obvious variations include

filout pretty ⟨text⟩ add!
filout ⟨text⟩ add ⟨vector⟩!
filout pretty ⟨text⟩ add ⟨vector⟩!

The *⟨vector⟩* could be given a name such as *list*:

☞list← ⟨vector⟩!

and then it is possible to type

filout ⟨text⟩ list!

will write out the definitions of objects named in *list.*

Or

filout ⟨text⟩ ☞list!

will first write out the definition of the vector *list* and then the definitions of the objects named in *list.* Variations with *pretty* and *add* are also possible.

Saving and Restoring Your Context

file ⟨text⟩ save!

will save your entire current state verbatim on the file ⟨text⟩.
Example:

file 'blockworld.sv' save!

Try

file 'dmt.boot' load!

to start the Interim Dynabook memory diagnostic.

file ⟨text⟩ load!

will restore you to the exact state when the file ⟨text⟩ was saved. Example:

file 'blockworld.sv' load!

This file is also one that you can *resume* from the operating system. That is:

resume blockword.sv ⟨return⟩

will restore you to the exact state when the file was saved.

Utilities

are already written programs which provide useful services such as reading the keyboard and the mouse, telling you how much room is available, and so forth.

nil

stands for the empty value in Smalltalk. It may be tested by saying:

null ⟨value⟩!

which will reply 1 if the ⟨value⟩ is *nil* (i.e., the empty value), and 'false' otherwise.

core!

will tell you how many words are left. Any reply smaller than 500 is courting disaster. If your space gets that low, or (worse) you get a diagnostic window with the message:

I've run out of memory

say:

expand ⟨number⟩!

This will remove ⟨number⟩ of scan lines from the screen and convert them to usable space at the rate of 32 words of space per scan line. So:

expand 100!

will increase your workspace by 3200 words.

addto ⟨classname⟩ ⟨vector⟩!

will add a definition whose meaning is ⟨vector⟩ to the class whose name is ⟨classname⟩.

Example, after typing:

addto box ☞(◅move ⇒ (SELF redraw (☞x←:. ☞y←:.)))!

box will know how to *move*.

{ ⟨value⟩ ⟨value⟩ ... ⟨value⟩ }!

will construct a vector of the values found between the curly brackets.

stringof ⟨value⟩!

will convert the ⟨value⟩ into an instance of the class *string* only if ⟨value is an object that responds to the message *print*.

base8 ⟨integer⟩!

will construct an instance of class *string* containing the octal representation (unsigned) of ⟨integer⟩.

eq ⟨value⟩ ⟨value⟩!

compares two Smalltalk pointers.

Keyboard Keys

⟨return⟩

moves following text to a new line when typing in. Will otherwise be ignored.

⟨bs⟩

removes any previous character (including ⟨return⟩).

Reading the Keyboard

kbd

will wait until a character has been typed and then reply with the numeric code of the character which was typed after being passed through a table which assigns (basically) standard codes to the character.

To receive an uninterpreted version of a character, use:

TTY

which will wait for a character to be typed and then reply with an uninterpreted result.

Smalltalk will not lose typed characters if no program is listening. Instead they are held in an ordered buffer waiting for a program to use *TTY* or *kbd*. To find out if there are any characters in the buffer, use:

kbck

which replies 'not-false' if characters have been typed and 'false' otherwise. A typical use would be:

kbck ⇒ (↑char ← kbd)

which will only use *kbd* if there is already a character waiting, and will then save the new character in *char*.

read

will gather up a *vector* of Smalltalk code. It first sends a prompt Ω to the display. Everything you type until a ! will then be made into a *vector* which is sent back.

read of ⟨text⟩

is the same as *read* except that the characters are found in ⟨text⟩ rather than taken from the keyboard.

ev!

repeatedly evaluates the vector *(cr read eval print)*; will, in effect, give you another level of Smalltalk evaluation.

to ev (repeat (cr. read eval print))!

Over and over, it will do a carriage return, put out a prompt character Ω, wait for input terminated by a !, send the resulting *vector* the message *eval* to get Smalltalk to execute the *vector*, and, finally, give the result of evaluation the message *print* in order to show the reply back to the user. The loop is infinite but:

done! or *⟨ctrl⟩D*

will terminate it. Here is a fancier version which will tell you the current level of evaluation:

```
☞ level ← 1!
to ev
  (☞level ← level + 1.
   repeat (cr. ☞level print. sp. level print. sp. read eval print)
   ☞level ← level - 1.)!
```

Notice that if the last token in the message is a period, then the sequence is not unlike

```
☞ a ← read of '.'.      a is (.)
☞ a ← a eval!           a evaluates to nil.
a print!                nil prints as nothing
```

⟨shift⟩ ⟨esc⟩

creates a subwindow in the dialog window. Allows Smalltalk evaluation as in the dialog window. (In effect, evaluates *ev* in the subwindow). To return to main window, type

done! or *⟨ctrl⟩D*

Subwindows can be nested as long as there is space to create a window with height greater than the font height. When a subwindow is created, reading characters is suspended in the main window; a return to the main window returns you to the precise place you left off, for example, in the middle of typing some expression.

⟨ctrl⟩ (

does an evaluation of the next expression at the time the keyboard input is read. This gives you an opportunity to perform a computation and have the result be used in the main expression being typed.

Transferring Messages

apply ⟨name⟩

will send the current message (the one which was sent to the context we are currently in) to the object which has name ⟨name⟩. For example, suppose 'we' are called 'bogus' and have a number of things we can do. Somebody sends us the message:

bogus sq 100+50!

and we have a line:

sq ⇒ (apply square)

then *square* will be *applied* to the remainder of our message *100+50* so that it can pick up the value 150 and draw the square with sides 150 units long.

apply ⟨name⟩ to ⟨vector⟩

gets the object which has name ⟨name⟩ and sends it the message ⟨vector⟩. Example:

apply square to (150)!

will draw the square with sides 150 units long. The important thing here, of course, is that we can compute a message and then send it to Smalltalk.

apply ⟨name⟩ in ⟨value⟩

will send the current message to the object which has name ⟨name⟩ using dictionaries whose *vector* starts with context ⟨value⟩. For example, if you would like to evaluate the message using only "top level" names (ignoring the dynamic environment), then try:

apply mumble in GLOB!

apply ⟨name⟩ to ⟨vector⟩ in ⟨value⟩

is the fullblown *apply*.

evapply

has exactly the same meaning as *apply* except that it expects a ⟨message⟩ of some kind to be evaluated rather than a ⟨name⟩. Example:

evapply (a⟨b ⇒ ('abcdefg')(this is vector)) to (length)!

will reply with the length of either the *string 'abcdefg'*, or the *vector (this is vector)*, depending on the values of *a* and *b*.

The optional formats for *evapply* are the same as for *apply* as described above.

Display Utilities

disp

is the local name of any *dialog* window. It is an instance of the class *dispframe*. When the mouse activates the window, *disp* may be used to send messages to the window or to find out things about it.

disp's frmx!

will tell you the x position (upper left corner) of the frame.

indisp ⟨value⟩ ⟨message⟩!

will temporarily redefine *disp* to be <value> and evaluate the message in this new context. This is usually used when the message contains print or read routines which assume that they will be using a dispframe named *disp*. The routine is defined as

```
to indisp disp
    (☞disp ← :.
     ⇑ 8 eval) !
```

sp

will print a space character.

cr

will print a carriage return.

dsoff

turns off the display and speeds up Smalltalk by a factor of 2.

dson

turns the display on again.

redo <integer>

where <integer> is the number of transactions (images of the Interim Dynabook) back from where you are, will re-evaluate the message at transaction <integer>.

Control Utilities

repeat (...)

contents of () will be re-evaluated until a *done* is encountered (or you strike the escape key). The escape will be from the innermost loop in which the *done* is enclosed.

done

will cause the loop to be exited.

done with <value>

will cause the loop to be exited with the value <value>.

again

will restart the innermost loop in which the *again* resides.

for <atom> ← <number1> to <number2> by <number3> do ( )

an iteration control feature--will re-evaluate contents of *( )* until the value of the index <atom>, starting at <number1> and stepped by <number3> each time, exceeds <number2>.

if <value> then <message1> else <message2>

if the value of <value> is 'not-false', then evaluate <message1> and do not evaluate <message2>. Otherwise, evaluate <message2>, ignoring <message1>.

do <integer> (...)

the contents of *( )* will be re-executed <integer> times.

Mouse Utilities

mx

replies with the horizontal position of the mouse. 0 is at the left margin, 512 is the right margin.

my

replies with the vertical position of the mouse. 0 is at the top of the screen, 808 is at the absolute bottom, 512 at the top and 680 at the bottom of the original dialog window.

mp

replies with an instance of class point such that mx = mp x, my = mp y.

button <numeric value between 0 and 7>

tests the mouse buttons singly and in combination.

button 0	'not-false' if no buttons are on
button 1	'not-false' if middle button is on (top is button nearest wire)
button 2	'not-false' if bottom button is on
button 3	'not-false' if bottom and middle are on
button 4	'not-false' if top button is on
button 5	'not-false' if top and middle button are on
button 6	'not-false' if top and middle are on
button 7	'not-false' if all the buttons are on

mem

mem loads integers from and stores them into real core. The important locations are:

clock

mem 0430	Read the clock
mem 0430 ← 0	Set the clock to zero

mouse

mem 0424	Read mouse x
mem 0425	Read mouse y
mem 0424 ← 0	Reset mouse x
mem 0425 ← 0	Reset mouse y

cursor

mem 0431	for i ← 1 to 16 (mem 0430 + i) are the cursor bits
for i ← 1 to 16 (mem 0430 + i ← shape[i])	Put new bits into cursor from vector named shape
mem 0105	Connections between mouse and cursor
mem 0105 ← 0	Disconnect cursor from mouse
mem 0426 ← x. mem 0427 ← y.	Move the cursor

interrupt character

mem 0107 ← 0177	Make DEL the interrupt character (instead of ESC)

display control block

mem 0420	Get pointer to display control block

keyboard, keyset, and mouse inputs

mem 0177034	Reads the first of 4 keyboard input words
mem 0177030	Reads the word with mouse and keyset bits.

Chapter V. EXAMPLE SMALLTALK CLASS DEFINITIONS

This chapter provides some examples of the use of various Smalltalk basic system classes and utilities. Included are samples of programming techniques as well as the contruction of new, interesting class definitions. The examples correspond to the basic classes defined in Chapter IV; they are presented in a "try it out" style with suggestions on problems and projects.

Arithmetic

Example: Figuring the Amortization of Loans

The problem we chose to demonstrate the use of *float* is the amortization of a loan in equal monthly payments. The main routine *payment* requests values for the loan principal, loan interest, number of years to pay off the loan, and the number of payments per year. It then carries out the following computation:

Let

rate = interest rate/ (100 * number of payments per year).

Let

increase = (1 + rate) raised to the power (number of years to pay off the loan * number of payments per year).

Then each

monthly payment = (amount of the loan * rate * increase) / (increase - 1).

The

total amount paid over the period =
(number of years to pay off the loan * number of payments per year) * monthly payment.

To report the results of the calculations, we need a reporting routine where we might say.

report 'Interest Rate as a Percentage is ' rate!

and expect to see

Interest Rate as a Percentage is 54.

The Smalltalk definition is

```
to report
(cr.                          Print a carriage return.
 disp ← :.                    Print the textual message in the dispframe.
 (:) print)!                  Print the value received in the dispframe.
```

Next we need to be able to receive the values from the keyboard for the parameters: number of years, rate, etc. We can use the Smalltalk utility *read.*

read will gather up a *vector* of Smalltalk tokens. It first sends a prompt 🕮 to the display. Everything you type until a ! will then be made into a *vector* which is sent back. For example, the result of saying:

☞*a ← read*!

and then typing:

do 4 (☺ go 100 turn 90)!

will associate the literal vector *(do 4 (☺ go 100 turn 90))* with the name *a*. If we say:

a!

the following will be the reply:

(do 4 (☺ go 100 turn 90))

If you send the message *eval* to *a*, Smalltalk will evaluate its contents:

a eval!

and a square will be drawn. To select the second element of the *vector a*:

a[2]!
4

To select the fourth element of the third element:

a[3][4]!
turn

Vectors have many capabilities. To see more, take a look at the definition of *vector* in Chapter IV.

read of <*text*> is the same as *read* except that the characters are found in <text> rather than taken from the keyboard. To help get values from the keyboard, you might define:

```
to demand  nm
 (☞nm ← ⦂.
   (◉as ⇒ (disp←:) nm print)
   ⇑nm ← read eval)!
```

Try it with:

demand spd as 'I want a new speed ' !
I want a new speed Ω*367*!

Then type:

spd!
367

or, without a specific message:

demand angle!
angle Ω*59*!

angle!
59

We will also need a method for converting the floating point numbers to nearest whole dollar notation. We can send the message $ to members of the class *float* and receive the value rounded to the nearest dollar.

```
addto float ☞ (↞ $ ⇒ (⇑  0.0 + (0 +  (0.5 + SELF) * 100)  / 100))!
```

The class float can now take a floating point number and round to the nearest dollar.

Now for the definition of *payment*, a method for computing the total amount of dollars paid on a loan at the end of the load period.

The Definition of the Class Payment

```
to payment principal interest period
      payments rate increase total                          Request four values
  (demand principal as 'Amount of the Loan in decimal d--d.dd '.
   demand interest as 'Interest Rate as a percentage'.
   demand period as 'Number of Years to Pay Off the Loan '.
   demand payments as 'Number of Payments per Year'.

                                                            Compute the rate, adding
   ☞ rate ←(0.0 + interest)  / 100 * payments.              0.0 to guarantee floating
                                                            point number.
                                                            Compute the increase.
   ☞ increase ← (1.0 + rate) ipow (period * payments).

                                                            Compute the total amount
                                                            paid over the period to the
                                                            nearest dollar.
   ☞ amount ← ((principal * rate * increase) / (increase - 1)) $.

   report 'Each Payment is $ ' amount.                      and report it.
                                                            Compute and tell total
                                                            amount paid over the period
   report 'Total Amount Paid is  $' ☞total ← amount * (period * payments).

                                                            Compute and
   report 'Total Interest Paid $ ' total - principal.)!     tell total interest paid.
```

Sample Interaction

Run this by typing

```
payment !
```

For example, the interaction between the user and *payment* might look like

```
Amount of the Loan in decimal d--d.dd Ω 30000.00!
Interest Rate as a percentage Ω9!
Number of Years to Pay Off the Loan Ω 30!
Number of Payments per Year Ω12!

Each Payment is $ 241.0
Total Amount Paid is $ 86760.0
Total Interest Paid is $ 56760.0
```

Sequential Dictionaries

include the classes: *vector, string, obset, stream, file.*

Stream.

Stringof is a method for converting a non-string value to a string. It is included in the basic Smalltalk system.

```
to stringof n
  (☞n ← :.
    ⇑ indisp stream (n print. ⇑ disp contents))!
```

Recall we have already defined *indisp* as

```
to indisp disp
  (☞disp ← :.
    ⇑ 8 eval)!
```

n is a value that we would like converted to a string. The simplest way to do this is to assume that the print method for any class is to convert its printable form into a string that it can send to *disp* (the generic name for a text display frame). We use *indisp* to set up a context in which *disp* is an instance of the class *stream*. We then send *n* the message *print* which should basically do: *disp* ← ⟨*string form of n*⟩. Since *disp* is a *stream*, it will store the string form as its contents, which we return as the proper reply.

Files.

The following routines (*xfer, copym, xplot*) are examples of the use of the class *file*. Each is a useful utility to have around.

(1) *xfer*

copies a single file. It is useful mainly for transferring files between disks on an Interim Dynabook with two disk drives. For example

```
xfer dp1 file 'valuable' old to dp0 file 'valuable' new!
```

copies a file named 'valuable' from disk 1 onto a newly created file of the same name on disk 0. To obtain this object type

```
filin 'xfer.'!
```

The definition is

```
to xfer f g h i
  (dsoff.                                   Turn off the display.
   ☞f ← :.                                  Fetch the instance of a file.
    (◂to ⇒ (☞ g ← :.                        The message (to) must appear for the format to be correct.

            repeat (g ← f's (numch = 512 ⇒ (sadr) sadr[1 to numch]).
             f's nextp = 0 ⇒ (done) f set to f's pagen + 1 0).
                                            The repeat-loop is copying each page of the file.
            g shorten to here)              Sets the pagen and bytec for g.
     ☞ g ← :.                               Otherwise say format is incorrect
     disp ← 'proper format is:
'.
     disp ← 'xfer ‹file› to ‹file›!
'.
     disp ← 'where ‹file› may be preceded by dp0 or dp1')

   f close. g close. dson.)!                Close the two files and turn the display on.
```

(2) *copym*

copies multiple files from one directory to a directory on the same or on another disk. For example, type

```
copym dp1 to dp0 ☞('file1') '.sr' !
```

This copies 'file1' from disk 1 to 'file1.sr' on disk 0 (the new file). The complete syntax for *copym* is

```
copym ‹source directory› to ‹destination directory› ‹vector of file names› ‹text›!
```

where ‹text› is the extension for the files on the destination directory. The extension is optional. The definition uses the object *xfer*.

```
to copym sourcedir destdir filenames ext i
  (☞sourcedir ← :.
   ◂to.
   ☞destdir ← :.
   ☞filenames ← :.
   (null (☞ext ← :.) ⇒ (☞ext ← ''))
   for i to filenames length - 1
       (xfer sourcedir file filenames[i] old to destdir file filenames[i] + ext new) )!
```

(3) *xplot*

xplot writes a screen image (bitmap) onto a file (86-87 disk pages, takes about one minute) for printing on an XGP with the XPLOT program. (Hence this is particularly useful to those readers with these facilities.) Either low or high resolution screen images can be plotted, but not both; i.e., only the low resolution (picture) part of a screen with both low and high resolution parts will be saved. Type

```
filin 'xplot.'!
```

The following definition requires the class *AREA* which is also provided below. Note *AREA* is a simple form of class *rectangle* defined in Chapter II and later in this chapter. The response of an instance of *AREA* to the message *makebuff* is a string containing the sequence of bits in the rectangular area. The file *'xplot'* also includes the objects *BLT*, *PNT*, and *bringitin*. This last one is a method for restoring a display screen from a file written by the object *xplot*. It expects one message--the file name.

```
to AREA a b c | origin extent
(◁'s ⇒ (⇑ 8 eval)
 ◁ is ⇒ (ISIT eval)
 ◁ makebuff ⇒ (☞a ← string 2 * extent y * ☞ b ← (extent x + 15) / 16.
               ☞ c← PNT a.
               BLT c + 2 b 0 extent x 0 extent y 0 mem 60 32 origin x origin y extent y 0.
               ⇑ a)
 isnew ⇒ (☞origin ← :. ☞ extent ← :))!

to PNT (mem 255 ← :. ⇑ mem 255)!


to xplot  f h i r s w
  (((☞f ← :) is file ⇒ ()
          ☞f ← file f ⇒() ⇑false).          Make sure f is a file.

    ☞w ← 255 ▯mem (☞h ← mem 272)+ 1.        Number of words per scan line.

    (0 < mem h + 1 ⇒ (f next word ← 2.      High resolution -- enlargement.
                      ☞s ← 2 * mem h + 3)   Number of scan lines.
      f next word ← 4. ☞s ← mem h + 3).     Low resolution.

    dsoff.                                  The screen area is written out on the file
    ☞ r ← AREA point 0 0 point w*16 1.      each time in the next loop.
    do 4 (f next word ← 0).                 Default values.

    for i to s                              -word count followed by bits in scan line.
        (f next word ← - w.  f ← r makebuff.
         r's (☞origin ← point 0 i)).        Move the area down the screen.
    f close.  dson. )!
```

Dispframe

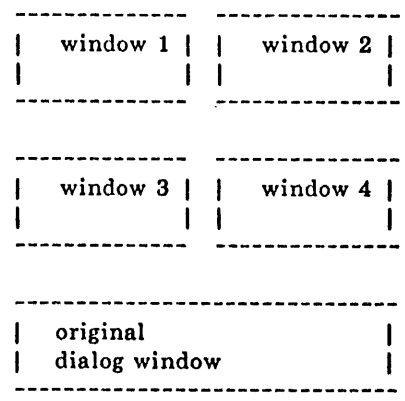
disp. As explained in Chapter III, *disp* is the local name of any dialog window. It is, in fact, an instance of the class *dispframe* and is created as

☞disp ← dispframe 16 480 415 168 string 520!

A Smalltalk window, as demonstrated in Chapter I, owns an instance of a *dispframe* whose name is *disp*. This particular name must be used because the Smalltalk read method assumes that all keyboard interactions will be carried out by displaying the typed characters in an instance of *dispframe* named *disp*.

As an example both of using this generic name as well as of using the four display routines (*dclear, dcomp, dmove,* and *dmovec*), try the following sequence.

1. Create four new windows on your display screen.
2. Place them in four quadrants of the screen, enlarging them to fill the area above the original dialog window.



3. Place the mouse cursor in window 1 and type *☞turt1 ← turtle frame disp!*
   This creates a turtle who lives only in this first window. *home* for *turt1* is the center of the window.
4. Repeat the above process: enter each of the remaining three windows and create turtles *turt2, turt3,* and *turt4.*
5. Now point in the original dialog window and try:

turt1 home erase! Note only window 1 is erased.
for i to 200 (turt1 go i turn 89)! Note the turtle draws lines only in its own window.
turt1's frame's (dcomp frmx frmwd frmy frmht)!
Complement window 1.

Try different designs in each of the four windows. Or try

turt2's frame's (dclear frmx frmwd frmy frmht 13107)!
turt3's frame's (dclear frmx frmwd frmy frmht 12121)!

As examples of using *dmove*, try making window 4 small and then

```
to mover
    (turt4's frame's (dmove frmx frmwd frmy frmht ☞frmx←frmx-5 ☞frmy←frmy-5 :))!
```

```
do 10 (mover 0)!
```

turt4's window moves toward the bottom left corner, replacing any information already displayed in the areas.

or

```
do 10 (mover 1)!
```

turt's window moves toward the bottom left corner, interacting with any information already displayed in the area.

Point class

This data type is used to design the class *rectangle* which can compute areas of intersection between two rectangles and create the rectangle that encloses two rectangles. An abbreviated version of the class *rectangle* was introduced at the end of Chapter II section on Paint Brush. To obtain this definition of *rectangle*, type

filin 'xyfns.'!

joe ← rectangle point 100 100 point 150 150!

Rectangle at upper left corner 100,100 and lower right corner 250, 250

joe has point 120 105!
point 100 100

That is, 'not false' and therefore true

joe comp!

Complements joe's bits.

joe clear -1!

Clears joe to all black.

joe clear 21212!
joe clear 052525!

Some nice patterns.

joe intersect jim ← rectangle point 140 120 point 150 170!

jim is a rectangle at upper left 140,120 and lower right 290,290. Reply is intersection of joe and jim, a rectangle at upper left 140,120 and lower right 250, 250 (origin is point 140 120; extent is point 110 130).

joe include jim!

Creates rectangle around joe and jim.

joe moveto 200 300!

Upper left corner is moved to 200, 300.

joe frame!

Draw a black border around the rectangular area.

The code for the class *rectangle* and some useful routines follow. Note two messages (*makebuff* and *loadbuff*) used in the definition of *AREA* as stored on file *'xplot'* could be included as messages understood by a *rectangle.*

```
to rectangle  a b c | origin extent
  (◀has⇒                                          Is a point inside rectangle?
    (☞c ← :.  ⇑origin  c  origin + extent)
   ◀'s⇒(⇑ 8 eval)
   ◀comp⇒
    (dcomp origin x extent x origin y extent y)
   ◀clear⇒                                        Expects bit patterns as a message
    (dclear origin x extent x origin y extent y :)
   ◀intersect⇒
    (☞c ← :.                                      Creates a rectangle that is the intersection of c
     ☞a ← origin max c's origin.                  and SELF if they have common area
      ☞b ←(origin + extent) min c's(origin + extent).  else, 'false'.
     a ≤ b⇒  (⇑rectangle a b - a) ⇑false)
   ◀include⇒                                      Creates rectangle around SELF and c.
    (☞c ← :.
     ☞a ← origin min c's origin.
     ☞b ←(origin + extent) max c's(origin + extent).
     ⇑rectangle a b - a)
   ◀moveto ⇒ (☞origin ← :)                        Move origin to a new point.
   ◀frame ⇒                                       Turtles understand how to go to a point as
     (☞a ← turtle.                                well as two numeric coordinates.
      a penup goto origin turn 90 pendn's width ← 2.
      a penup goto origin turn 90 pendn's width ← 2.
     do 2  (a go extent x turn 90 go extent y turn 90))
   ◀is ⇒(ISIT eval)
   ◀print ⇒
    (☞rectangle print sp origin print sp extent print)
   ◀ paint ⇒ (CODE 41)                            This message was discussed in Chapter II
                                                  section on Paint Brush.
   isnew ⇒  (☞origin ← :.  ☞extent ← :))!

to waitnext  x
  (☞x ← 8.                                        Stay in this routine until
   repeat (x eval ⇒ ( ) done)                     x is first 'false' and then finally
   repeat (x eval ⇒ (done)))!                     'not-false' again.

 to bug                                           Wait to get the mouse point
      (waitnext but1on. ⇑mp)!                     when button 1 is pressed.
```

A demonstration to try often is

```
to xydemo ← class a b c
  (☞a ← rectangle ☞b ← bug bug - b.
   a comp.
   ☞c ← rectangle ☞b ← bug bug - b.
   c comp.
   ☞b ← a intersect c.
   (b⇒(b clear 13107))
   (a include c) frame.)!
```

Type

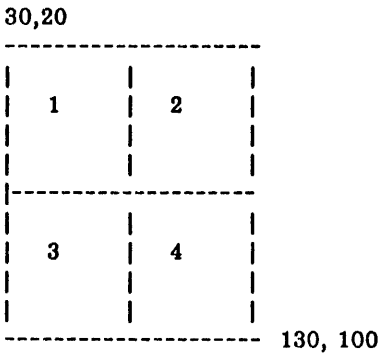
```
xydemo!
```

The result of pointing to different screen locations is a geometric design formed by the interaction of black, white, and gray rectangles.

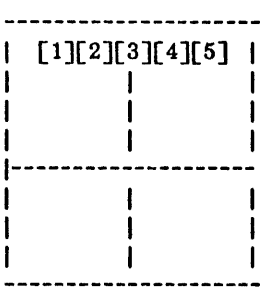
Dictionary of Areas and Points

An Obscure Challenge for the Day: when does this blow up?

Suppose the screen is divided into a main area that is a rectangle point 30 20 point 100 80; and the subareas within the main area are 50 wide and 40 high. There are four such subareas. The purpose of a dictionary of points on the screen is to be able to designate areas as menu locations or nodes of a tree or whatever, and to be able to recognize, quickly, in which area the mouse is located.



Suppose we create a menu that has five menu squares (1, 2, 3, 4, 5) located in subareas 1 and 2. Each menu square has length 14 units. Further, suppose the upper left corner of the first square is point 45 30.



Then, we have

☞*dictionary ← xydic 50 40 in rectangle point 30 20 point 100 80*!

Create the main area and subareas.

☞*menu ← vector 5*!
for i to 5 do
*(dictionary ← menu[i] ← rectangle point 45+(i-1)*14 30 point 14 14)*!

Store menu squares 1 - 5.

dictionary print!
3 in area 1
3 in area 2
0 in area 3
0 in area 4

Print number of items in each subarea followed by the subarea index.

dictionary map ☞(comp)!

Tell all the menu squares to complement.

dictionary index point 50 70! *3*	Given a point (50 70), compute in which subarea it falls.
dictionary find mp!	Given a point (mouse point), ask all the stored areas if they have the point. Return the first one that says yes.
dictionary delete menu[3]!	Delete the third menu square.
dictionary edit (delete) menu[3]!	Editing method used by messages delete and ←.

A file exists on the basic Smalltalk disk that contains the following definition. Type

filin 'xydic'!

to try out this dictionary method.

```
to xydic exp i input p val | all areas brect ncols xsize ysize

(◀index ⇒ (☞p ← :.
          ⇑1 +((p x - brect's origin x) / xsize) + ncols * (p y - brect's origin y) / ysize)

 ◀find ⇒ (☞p ← :.
          brect has p⇒ (☞val ← nil.
                        areas[SELF index p] map ☞
                          (vec[i] has p ⇒ (done with ☞val ← vec[i])).
                        ⇑val)
          ⇑false)

 ◀edit ⇒ (☞exp ← vecmod 8 2 0 ☞input ← :.
          ☞val ← (SELF index input frame's(origin + point extent x 0))
                    - ☞i ← SELF index input frame's origin.
          for i ← i to SELF index input frame's(origin + point 0 extent y) by ncols
                (for p ← i to i + val (evapply areas[p] to exp)).
          apply all to exp)

 ◀← ⇒ (SELF edit (←) :)

 ◀delete ⇒ (SELF edit (delete) :)

 ◀map ⇒ (all map :)

 isnew⇒ (☞xsize ← :. ☞ysize ← :.
         ☞brect ← (◀in⇒(:) rectangle point 0 0 point 512 512).
         ☞ncols ← brect's extent x / xsize.
         ☞areas ← vector ncols * brect's extent y / ysize.
         for p to areas length (areas[p] ← obset).
         ☞all ← obset)

 ◀print ⇒ (for p to areas length (areas[p] length print. sp.
           disp ← ' in area '. p print. cr)))!
```

Turtles

Try

```
☞turt ← turtle frame dispframe 16 100 16 100 " .!
turt home erase.!
for i to 300 (turt go i turn 89).!
turt's frame's (dcomp frmx frmwd frmy frmht).!
```

The first three statements create a turtle in a 100 by 100 rectangular area in the upper left portion of the display screen, clear that area to white and draw a spiral using black lines. The last statement enters the context of the turtle's display frame in order to use the frame boundary parameters in order to complement the area (white to black, black to white).

To sketch with characters or text, try

```
to draw turt t

  (☞turt ← turtle.                    Create a drawing turtle.
   turt home xor turn 90.             Painting is different if the ink
                                      is black or white.
   ☞ t ← :.                           Fetch the "paint brush"

   repeat (button 4 ⇒ (turt penup goto mp pendn ← t)
           button 2 ⇒ (done)))!

draw '☺'!                             Paint with "smiley"

draw 'hello'!                         or the text 'hello'

draw 97!                              or the character 'a'.
```

Designing your own character is another way to design a paint brush!

Commander Turtle

Here is a nice way to distribute turtle messages to more than one turtle at a time. The idea is to create a "commander" turtle. Any messages he receives, he sends on to all the members of his troop.

```
☞joe ← commander 4!                   joe commands a troop of 4 turtles. Each turtle moves to the center (home)
                                      of the display area. Then joe sends himself the message fan.
joe go 100!                           Each member of the troop moves <number> of units.
joe turn <number>!                    Each member of the troop turns <number> of units.
joe penup!                            Each member of the troop picks its pen up.
joe pendn!                            Each member of the troop put its pen down.
joe home!                             Each member of the troop moves to the center of the display area.
joe fan!                              Each member of the troop turns in a unique direction and changes ink color
                                      such that member i has ink color i+1.
joe's ink ← <integer>!                Set the ink color of each member to <integer>.
joe's width ← <integer>!              Set the width of each member to <integer>.
```

Try

```
☞☺ ← commander 4!
dragon 6!
```

Recall the definition of dragon in Chapter II sends messages to ☺. Here, ☺ is no longer a turtle, but a turtle commander.

to see four dragon curves draw on the screen. For those curious, we include the class definition. Note the use of colored ink assumes a color version of Smalltalk. The dispframe *colorframe* is defined as

```
☞colorframe ← dispframe 0 256 0 128 ".
```

```
to commander a b | turts

(◀ go      ⇒ (☞a ← :. turts map ☞(go a).  ⇑SELF)
 ◀ turn    ⇒ (☞a ← :.  turts map ☞(turn a).  ⇑SELF)
 ◀ penup   ⇒ (turts map ☞(penup). ⇑SELF)
 ◀ pendn   ⇒ (turts map ☞(pendn). ⇑SELF)
 ◀ home    ⇒ (turts map ☞(home). SELF pendn. ⇑SELF)
 ◀ fan     ⇒ (for a to turts length do
              (turts[a] turn(a - 1) * 360 / turts length.
              turts[a]'s ink ← a + 1).
             ⇑SELF)
 ◀ 's      ⇒ (◀ink ⇒ (◀←.  ☞a ← :.
               turts map ☞ (vec[i]'s ink ← a))
              ◀width.  ◀←.  ☞a ← :.
              turts map ☞('s width ← a).)
 isnew     ⇒ (☞a ← :.
              ☞turts ← vector a.
              for b to a (turts[b] ← turtle frame colorframe)
              SELF home fan))!
```

Control Classes for Repetition and Alternate Paths

repeat, do, for, if

The usual methods for repeatedly evaluating an expression use one of three routines already presented: *repeat, for*, and *do* . The method of *for* can be defined as

```
to for step stop var start exp
      (↑var ← 8.
       ↑start ← (« ← ⇒ (:) 1)
       ↑stop  ← (« to ⇒ (:) start)
       ↑step  ← (« by ⇒(:) 1)
       «do.                                  do is optional
       ↑exp ← 8.
       var ← start.
       repeat ((step > 0 ⇒ (var eval > stop ⇒ (done))
                    var eval < stop ⇒ (done))
                 exp eval. var ← (var eval) + step.))!
```

The form of a Smalltalk conditional statement, *if-clause ⇒ (then-clause) else-clause*, has also already been shown in many contexts. The Algol "if...then...else..." syntax can be achieved by defining *if* as follows.

```
to if exp
      ((↑exp ← :) ⇒ (« then ⇒ (↑exp ← :. «else ⇒ (8. exp) exp)
                     error ↑(no then ))
       « then ⇒ (8. «else ⇒ (↑exp ← :) false)
       error ↑(no then)) !
```

For example,

```
↑val ← if a > 10 then 4 else (if a < 10 then (-4) else 0)!
```

val will be 4, -4, or 0, depending on the value of *a*.

again

is a Smalltalk method for redoing the most recent *repeat, do* or *for* loops. It is one way of iterating on a given condition, while defaulting to end the loop. For example, suppose we send the message

```
↑set ← makelist mary or joe or henry!
```

expecting to form a list of alternatives terminating when no further alternatives exist.

```
to makelist list
      (↑list ← obset.
       repeat (list ← 8.                    Obsets form unions.
              «or ⇒ (again) done)           Continue if see word "or".
       ⇑ list)!                             Reply with the list.
```

while

A *while* clause lets us send messages of the form

```
↑str ← stream!
while (kbck and ((↑t ← kbd) ≠ 13))
    do (str ← t)!
```

That is, store keyboard strokes into the stream *str* as long as there is a character in the input buffer and the character typed is not a carriage return (whose Ascii code representation is 13). This definition is not part of the basic Smalltalk system.

```
to while Cond Exp
      (↑Cond ← [].
       ◁ do.                                          do is optional
       ↑Exp ← [].
        repeat (apply Boolean to Cond ⇒ (Exp eval) done))!


to Boolean result
      (↑result ← :.
        repeat (◁or ⇒ (result ⇒ ([]) ↑result ← :)
                ◁and ⇒ (result ⇒ (↑result ← :) [])      Right side of the and part will not
                                                        be evaluated if left part is 'false'.
                ↑ result))!
```

Zahn's Device

The following is an implemention of a simple "until-like" structure, very much like Zahn's original suggestion, which allows multiple exits from a loop [Zahn, A control statement for natural top-down structured programming, *Symposium on Prog. Languages*, Paris, 1974]. The intent was to be able to write in Smalltalk a minimal, event-driven keyboard/display routine like this one:

```
until CR or DEL do
      (↑t ← kbd.
       disp ← t.
       t = 13 ⇒ (CR)
       t = 127 ⇒ (DEL))
  case
      CR  : (disp ← 'normal exit.')
      DEL : (disp ← 'punt exit.')!
```

To implement this control structure in Smalltalk, a class of objects called *events* was defined such that each instance, when it is awakened, executes a piece of code and breaks out from a loop.

```
to until tempatom statement
      (repeat  (↑↑tempatom ← [].
                tempatom ← event.
                ◁ or ⇒ (again) done)
       (◁do ⇒ (↑↑statement ← []))
       (◁case ⇒ (repeat (↑↑tempatom ← [].
                         tempatom eval is event ⇒
                           (◁:. tempatom eval newcode [].)
                         done)))
        repeat (statement eval))!
```

```
to event | mycode
      (isnew ⇒ (☞mycode ← vector 3.
                mycode [2] ← ☞done.)
       ☟ newcode ⇒ (mycode[1] ← :.)
       ☟ is ⇒ (ISIT eval)
       mycode eval)!
```

Event is an example of constructing a vector of code that will be evaluated at some later time. When an instance of *event* receives the message *newcode*, it stores away some message as the first objects in the vector *mycode*. The last object is the message *done* which, when *mycode* is evaluated, forces a break out of the repeat loop in *until*. Hence, if we run the above example of using *until*, we have

```
☞tempatom ← 8.                    Pick up the word CR and store in tempatom.
tempatom ← event.                 CR is made an instance of the class event.
☟or                               We see or, so
⇒(again)                          go back, pick up DEL, and make it an instance of event.
done                              Now we are done.
☟do                               We see the word do.
☞statement ← 8                    Statement is the vector (☞t←kbd. ... ⇒(DEL).
☟case                             We see the word case.
repeat (☞tempatom←8.              We see the word CR again and store the name in tempatom.
tempatom eval is event            The value of tempatom is Cr, an event.
(☟:.                              We see colon, :, so we
tempatom eval newcode 8)          send the event CR the message newcode and pick up the code
                                  disp ← 'normal exit'. Do this again: pick up DEL and send it the
                                  message newcode, picking up code disp ← 'punt exit'.
done)                             There are no more case statement words so
repeat (statement eval)           repeatedly evaluate the vector (☞t←kbd...), an expression
                                  that will continually request keyboard input until that input is
                                  a carriage return or delete character in which case the
                                  corresponding event will be run in order to evaluate mycode.
                                  Evaluating mycode results in execution of a done message, hence
                                  terminating the repeat loop.
```

Case Statement

A method for simulating case statements in Smalltalk is to index into a vector of vectors or atoms that can be evaluated. The general message form is

```
⟨vector⟩ [ ⟨integer⟩ ] eval!
```

Such a case statement can be seen in the routine used to realize a display window move, delete, create, or grow depending on which window corner has the mouse cursor. The routine returns 'false' if the cursor is not in a corner. Note, in the statements below, the index = 1 + corner selected.

```
to frmedit disp
    (☞disp ← :.
      ☞((⇑false)                                  index =1, no corner selected
          (disp fclear. waitnext (but1on).        index=2, move
           disp frame white. disp moveto mx my.
           disp display)
          (1=sched vec length ⇒( ) disp hide.     index=3, delete
             sched delete task. done.)
          (contents copy)                         index=4, create
          (disp fclear. waitnext (but1on).        index=5, grow
             disp frame white. disp grow mx my.
             disp display))
          [ 1 + disp corner mx my] eval )!        index evaluation
```

For instance, if the mouse is in upper right hand corner of the display window, then

disp corner mx my = 2

Add 1 and we get and index of 3, picking out the code to delete the current window.

Scheduling Methods: sched and window

Recall that Smalltalk has a USER task which is continually evaluated. (See Chapter III section entitled *The User Task*).

One method useful for scheduling the display windows we have been working with is to replace the USER task with a request to send the message *run* to each item stored in an obset. We have chosen to name this obset *sched.*

```
PUT USER ☞DO ☞(sched map ☞(☞task ← vec[i].
                    apply task to ☞(run) in GLOB))!
```

or

```
PUT USER ☞DO ☞(sched map ☞(apply each to ☞(run) in GLOB))!
```

Suppose *sched* contains three items, each one an instance of the class *window* (we will examine the code for this class in a bit). Then, in sequence, the temporary variable *task* is set to the value of vec[1], vec[2], and vec[3] (the local bindings in *sched* for the three instances of *window*). Each value of *task* is sent the message *run.* This is a round robin method for scheduling objects, giving each object the opportunity to run if it so chooses. Each object stored in *sched* must be able to receive the message *run.*

A window that can be scheduled has two instance variables, an instance of the dispframe in which we expect to read and print any keyboard i/o, and an instance of a class that knows about and can edit the objects living in the dispframe. We will present three examples of this second kind of class: a Smalltalk dialog window (*stwindow*), a window for invoking the Smalltalk class editor (*edwindow*), and a picturewindow (*picturewindow*).

Window. The class *window* looks like

```
☞w ← window dispframe 10 100 10 50 string 50 ‹editor›!
```
Create a window in which the contents is defined as some editor.

```
w run!
```
This is the message we expect to send as part of the USER task.

```
w contents ‹message›!
```
window contents is ‹editor›. Send this ‹editor› the message ‹message›

```
to window | disp contents
    (☟run ⇒ (disp hasmouse ⇒
              (contents enter.
                repeat (disp hasmouse ⇒ (kbck ⇒ (contents kbd)
                                                0 < mouse 7 < ⇒ (contents bug)
                                                contents running)
                        done)
                contents exit))
    ☟contents ⇒ (⇑ apply contents)
    ☟is ⇒ (ISIT eval)
    ☟'s ⇒ (⇑ ♁ eval)
    isnew ⇒ (☞disp ← :. ☞contents ← :. contents new))!
```

The value of *disp* does not have to be a dispframe, but it does have to respond to the message *hasmouse.* Notice that the main method for sending a message to the object whose name is *contents* is to send it indirectly through the class *window.* When a *window* sees the message word *contents* it gives the object *contents* permission to examine the message. For example, if *contents* is an instance of *stwindow*, defined next, and we want to send that instance the message *running*, we could do so indirectly by typing

w contents running!

where *w* is an instance of *window* and the value of *w contents* is an instance of *stwindow.*

Smalltalk Dialog Window.

Now for some examples of <editor>, each of which must understand the messages sent to it by *window*: *enter, running, kbd, bug, exit, new.*

The particular method used to define stwindow says that the final action in creating an instance of the class is to return an instance of *window.* Hence it is not possible to send messages directly to instances of *stwindow*; it is only possible to send messages indirectly through the class *window.*

sched ← ☞*st* ← *stwindow dispframe 10 100 10 50 string 50*!

Create a Smalltalk window where the display area is initially at 10,10 with width 100 and height 50. Note that st is an instance of window, not stwindow. The value of st contents is the desired instance of stwindow.

st contents enter! Show the dispframe

st contents running! Blink the prompter.

st contents kbd! Read an expression from keyboard.

st contents bug! See where the mouse is pointing and take any appropriate actions.

st contents new! Print a message in the window.

st contents copy! Create another stwindow in st's own image.

```
to stwindow
  (◁enter ⇒ (disp display)
    ◁running ⇒ (disp ← 20. do 10 ( ). disp ← 8)      blink the prompt character in the window
    ◁kbd ⇒ (cr. read eval print sp)
    ◁bug ⇒ (frmedit disp)                            frmedit was defined previously.
    ◁exit ⇒ ( )
    ◁copy ⇒ (sched←stwindow newframe)                newframe creates dispframe in the upper left corner
                                                     of the display screen.
    ◁new ⇒ (disp ← 'A SMALLTALK window ')
    ◁is ⇒ (ISIT eval)
    ◁'s ⇒ (⇑ 8 eval)
    isnew ⇒ (⇑window (:) SELF))!
```

```
to newframe f

  (☞f ← dispframe 16 256 16 112 string 1000 font disp's (font).

   f's (☞winht ← frmht).

   ↑ f)!
```

Edit Window. The content of this window is a list of names of defined classes. Pointing at one of the names in the window invokes the Smalltalk class editor for the class. This is a useful utility for avoiding typing *edit ⟨name⟩*! The same method for defining the window is used here as was used in *stwindow*: the reply from *isnew* is an instance of *window*; messages to *edwindow* must be sent indirectly through *window*.

sched ← ☞edw ← edwindow!	edw is an instance of window; its instance variable contents is an instance of edwindow. The window's dispframe is newframe.
edw contents enter!	Display the dispframe.
edw contents running!	Blink a thick-lined square image (Ascii 4).
edw contents kbd!	Create a subwindow and call on ev. I.e., repeat (cr read eval print sp).
edw contents bug!	Check the four corners (copy does not work)...if mouse is not in corners find which name the mouse is pointing at and call on the editor for the appropriate class.
edw contents show!	Print the token 'edit:' followed by name stored in the vector.
edw contents exit!	Do nothing special.
edw contents new!	Display the dispframe.

```
to edwindow a i | setname

  (◅enter ⇒ (disp display)

   ◅running ⇒ (disp ← 4. do 10 ( ). disp ← 8)

   ◅kbd ⇒ (disp sub ☞(ev))

   ◅bug ⇒ (frmedit disp ⇒ ( )
          ☞i ← disp mfindt mx my [1].
          i⟨2 ⇒ ( ).
          ☞a ← (setname eval)[i-1].

          ☞i ← a eval.
          edit i.
          a = setname ⇒ (SELF show))

   ◅show ⇒ (disp clear. disp ← '
edit:'.
          ☞a ← setname eval.
          for i to a length - 1 (sp. a[i] print))
```

The word "edit:" adds a token to the count. value of a is the class name.
i is now a pointer to the class to be edited.

Print the token 'edit:' followed by the names in the atom a.

```
exit ⇒ ( )

new ⇒ (disp frame black. SELF show.)

is ⇒ (ISIT eval)

's ⇒ (⇑ 8 eval)

isnew ⇒ (☞setname ← 8.

        ⇑ window newframe SELF))!
```

Picture Window. This simple picture editor is an example of the use of a turtle "living" in a *dispframe*. It makes use of the class *point* as well as *obset* and *apply*.

```
sched ← ☞pw ← picturewindow 16 100 16 100 string 50!
```

Creates a window for sketching at location 16,16. window is 100 wide, 100 high. Again, pw is an instance of window, pw contents is an instance of picturewindow.

pw contents enter!	Show display frame and sketch.
pw contents running!	Do nothing special.
pw contents kbd!	Read the keyboard but do not evaluate expression.
pw contents bug!	Check four corners; otherwise, draw a line to the mouse point. If middle mouse button pressed, pick turtle pen up.
pw contents exit!	Do nothing special.
pw contents new!	Erase the display area.
pw contents sketch!	Draw lines between the points in the sketch unless point preceded by penup command.
pw contents copy!	copy has a new meaning: erase the sketch.

```
to picturewindow  var | df 😊 pics

  (◀enter ⇒ (df display.  SELF sketch)

   ◀running ⇒( )

   ◀kbd ⇒(cr read)

   ◀bug ⇒ (frmedit df⇒(SELF sketch)
            (pics vec length = 0 ⇒ (pics ← 👉penup. 😊 penup)
                                                  When first start pick pen up, or if
             button 1 ⇒ (pics add 👉penup. 😊 penup)
             😊 pendn)                            middle button pressed, pen up.
            pics ← 👉var ←(mp - (point df frmx df frmy)).
                                                  Find mouse point and store point relative to
                                                  the display window.
            😊 goto var)                          Draw the line.

   ◀exit ⇒ ( )

   ◀new ⇒ (😊 erase)

   ◀sketch ⇒ (pics vec length = 0⇒( ).          Nothing to sketch. Should pen be up?
              pics map 👉 (👉penup = vec[i]⇒(😊 penup)
              😊 goto vec[i] pendn))             Draw line to the point.

   ◀copy ⇒ (df clear. 👉pics ← obset)           Delete sketch points.

   ◀'s⇒ (⇑ 8 eval)

   ◀is ⇒ (ISIT eval)

   isnew ⇒ (👉df ←(apply dispframe).            Instance of dispframe created by receiving
                                                values from picturewindow's message.
             👉😊 ← turtle frame df.             Turtle lives in this new frame.
             👉pics ← obset.                    Sketch points stored in an obset.
             ⇑window df SELF))!                 Create the window.
```

Loopless Scheduling

The following is an attempt to select some conventions for scheduling classes, while minimizing, if not eliminating, the use of explicit *repeat* or *for* loops. We define *startup*, a method for waking up each class instance and giving each a chance to grab control and remain in control until some quit condition becomes true.

```
to startup task
  (☞task ← :.
   (◁in ⇒ (☞GLOB ← :))                    Define context for evaluation.
    task startif ⇒ (task firsttime.       Task starts, send firsttime.
                    repeat(task quitif ⇒ (done)
                    task eachtime)        Keep sending message eachtime
                                          until quitif returns 'not-false' value.
                   ⇑task lasttime)        Finally send message lasttime.
    ⇑false)!
```

We will still use *sched* to hold the scheduled objects. The USER task is

```
PUT USER ☞DO ☞(sched map ☞(startup each in GLOB))!
```

A task may choose to start, for example, if mouse cursor is in particular location or mouse buttons are pressed or objects are waiting in a queue. The first time the task runs it may want to clean up some graphic information or set a timer or take first object out of the queue. A task may decide to quit if some clock timer has run out or the mouse is no longer in the correct position. Each time a task runs, it takes whatever actions are appropriate; for example, the window might check to see if a mouse button is pressed and the mouse cursor is in one of the corners. Hence, by convention, a scheduled object must respond to *startif, firsttime, quitif, eachtime, lasttime.* So that no errors occur if an object does not respond to these messages, we initialize things with

```
☞startif ← ☞firsttime ← ☞eachtime ← ☞lasttime ← nil.
to quitif (⇑false)!
```

The class *window* which acted as a task master before is no longer needed. Methods for blinking the prompter and waiting for an expression to evaluate true (*waitnext*) can be (re)defined. The class *prompt* simply sets a timer, displays the prompt character and does nothing until the timer runs out at which time it backspaces to erase the image. When *prompt* is the only scheduled object, we see a blinking prompt character.

```
to prompt | t
  (◁firsttime ⇒ (disp ← 20)              Show Interim Dynabook image.
   ◁quitif ⇒ (⇑t < mem 280)              mem 280 is the clock.
   ◁lasttime ⇒ (disp ← 8)                Print backspace to erase image.
   isnew ⇒ (☞t ← 10 + mem 280))!         Set timer.
```

The next object, *waitnext*, also ignores some of the messages.

```
to waitnext | notoffyet expr
  (◁quitif ⇒ (notoffyet ⇒ (⇑expr eval is false) ⇑ expr eval)
   isnew ⇒ (☞expr ← 8. ☞notoffyet ← true. startup SELF.
            ☞notoffyet ← false. startup SELF))!
```

The object *frmedit* is almost the same. The only exception is index 4 which originally was *(contents copy)* but now must be the actions previously taken by *(contents copy)*. In the case of *stwindow*, this should be *(sched ← stwindow newframe)*. But *edwindow* wants to do nothing and *picturewindow*

wants to say *(disp clear. ☞pics ← obset)*. Alternatives are to write separate *frmedit* routines or to send the code as a message to be evaluated at a later time. We will use this last idea.

```
to frmedit disp expr
 (☞disp ← :. ☞expr ← nil. ☞expr ← 8.
   ☞((⇑false)
       (disp fclear. waitnext (but1on).
        disp frame white. disp moveto mx my. disp display)
       (1=sched vec length ⇒( ) disp hide. sched delete task. done.)
       (expr eval)
       (disp fclear. waitnext (but1on).
        disp frame white. disp growto mx my. disp display))
              [ 1 + disp corner mx my] eval )!
```

The Smalltalk dialog window is now defined as

```
to stwindow | disp
 (◁startif ⇒ (⇑disp hasmouse)
  ◁firsttime ⇒ (disp display)
  ◁quitif ⇒ (⇑disp hasmouse is false)
  ◁eachtime ⇒ (kbck ⇒ (cr read eval print sp)
               0 < mouse 7 ⇒ (frmedit disp (sched ← stwindow newframe))
               startup prompt)
  ◁is ⇒ (ISIT eval)
  ◁'s⇒ (⇑ 8 eval)
  isnew ⇒ (☞disp ← :. disp clear.  disp ← 'SMALLTALK at your service ' ))!
```

edwindow and *picturewindow* can be defined as

```
to edwindow a i | setname disp
 (◁startif ⇒ (⇑disp hasmouse)
  ◁firsttime ⇒ (disp display)
  ◁quitif ⇒ (⇑disp hasmouse is false)
  ◁eachtime ⇒ (kbck ⇒ (disp sub ☞(ev))
               0 < mouse 7 ⇒
                 (frmedit disp ( ) ⇒ ( )
                  ☞i ← disp mfindt mx my [1].
                  i<2 ⇒ ( ).
                  ☞a ← (setname eval)[i-1].
                  ☞i ← a eval.
                  edit i.
                  a = setname ⇒ (SELF show))
               startup prompt)
  ◁show ⇒ (disp clear.  disp ← 'edit: '
           ☞a ← setname eval.
           for i to a length - 1  (sp.  a[i] print))
  ◁is ⇒ (ISIT eval)
  ◁'s⇒ (⇑ 8 eval)
  isnew ⇒ (☞setname ← 8.
           ☞disp ← dispframe 16 256 16 112 string 1000.
           disp clear. SELF show))!
```

The prompt character is different.

```
to picturewindow | df 😊 pics
 (◀startif ⇒ (⇑df hasmouse)
  ◀firsttime ⇒ (df display. SELF sketch)
  ◀quitif ⇒ (⇑df hasmouse is false)
  ◀eachtime ⇒ (kbck ⇒ (cr read)
               0 < mouse 7 ⇒
                (frmedit df (df clear. 👉pics←obset) ⇒ (SELF sketch)
                 (pics vec length = 0 ⇒ (pics ← 👉penup. 😊 penup)
                 1 = mouse 7 ⇒ (pics add 👉penup. 😊 penup)
                 😊 pendn)
                pics ← 👉var ←(mp - (point df frmx df frmy)).
                😊 goto var))
  ◀sketch ⇒ (pics vec length = 0⇒( ).
             pics map 👉 (👉penup = vec[i]⇒(😊 penup)
             😊 goto vec[i] pendn))
  ◀ is ⇒ (ISIT eval)
  ◀'s⇒ (⇑ ⁸₈ eval)
  isnew ⇒ (👉df ← apply dispframe.
           👉😊 ← turtle frame df.
           👉pics ← obset.
           😊 erase. df display))!
```

Messages can now be sent directly to instances of *stwindow*, *edwindow*, and *picturewindow*.

A Sample Text Editor

The purpose of this example is to demonstrate text management within a display frame (*dispframe*)--how to

i. display text
ii. use mouse for pointing, keyboard for editing (or alternatively, set up an editing menu such as in the Smalltalk editor)
iii. manipulate the text

Insert, delete, replace and append text can be accomplished with insert only:

action	interpretation
point someplace and start typing	insert, append
point to subset of the text and start typing characters	replace
point to subset and type 'del'	delete

Note: when typing, will handle backspace (bs); If (<doit>) as character, not as terminator; and delete (del) key.

A paragraph has some area on the display screen, is framed, and does not scroll unless it reaches the bottom of the screen.

Call it *pdisp*.
pdisp is a *dispframe*.
The window height of *pdisp* (*winht*) should extend from the upper left corner to the bottom of the display screen in order to avoid scrolling.
The frame height of *pdisp* (*frmht*) should indicate bottom of last line of text.

fontheight ← 14.
pdisp ← dispframe 0 1 0 fontheight string 0 noframe.

A paragraph contains some text.

Call it *buf*.
buf is a string.
There is a pointer to the last character in *buf*.
Call it *last*.
last is a *number*.
These correspond to instance variables in a *dispframe* but paragraph wants local manipulative control of the textual information.
We can give *buf* a textual value when we create the instance.

buf ← (◀of ⇒ (:) string 0).
last ← buf length.

A paragraph contains pointers into a subset of the text.

Call the points *p1* and *p2*.
p1 and *p2* are each instances of the class *point*.
They indicate the beginning and ending of a selected subset of text.
These points correspond to indices into the text string
Call the indices *loc1* and *loc2*.

↑loc1←↑loc2←buf length.!

A paragraph has the selected subset of text complemented to provide graphic feedback.

Assume there is a class, *dfcomp*, owned by the paragraph class to perform the complementation from *p1* to *p2* within the *dispframe.*

dfcomp pdisp p1 p2!

A paragraph's text can be manipulated.

(1)	Show correct text	tell *pdisp* to show *buf[1 to last]*
(2)	Select an area of text	start with mouse button press in order to select space between characters = p1; hold down button to pick up characters dynamically and then release the button. The final mouse position = p2
(3)	Replace selected text by new text	start typing if 'del' and loc1 not same as loc2, then delete selected text otherwise delete selected text and replace with keyboard input otherwise, keyboard input replaces selected text.
(4)	Might want to give the paragraph a name and store/retrieve it on a disk file	

A solution to the text complement problem for a dispframe

Assume have two points indicating beginning and ending of line of text

p1	beginning point
p2	ending point
df	dispframe

If p1 and p2 are the same point, complement nothing

If p1 is lower in the dispframe than is p2, complement nothing or reevaluate the routine, changing roles of p1 and p2

If p1 is higher in the dispframe than is p2:

```
        p1----------
--------------------
--------------------
------------p2
```

complement from p1 to p2 requires possibly three parts

(1) complement first line starting at p1
(2) complement full middle lines
(3) complement last line up to p2

Since the last line may be the first line p1---------p2,
(3) is solved by *dcomp p1 x (p2 x - p1 x) p2 y fontheight.*

(1) is needed if p1 y < p2 y; it is solved by
dcomp p1 x (df (frmx + frmwd) - p1 x) p1 y fontheight.

If we then redefine p1 as

☞p1 ← point df frmx p1 y + fontheight.

we set p1 at the beginning of the second line.
If now p1 and p2 are at same height and therefore same line, (3) solves it.
Otherwise, (2) is needed to fill middle lines by

dcomp p1 x (df frmwd) p1 y (p2 y - p1 y).

Putting this together we have

```
to dfcomp  df p1 p2
  (☞df ← :.
   ☞p1 ← :.
   ☞p2 ← :.
   (p1 y < p2 y ⇒ (dcomp p1 x (df (frmx + frmwd) - p1 x) p1 y fontheight.
                   ☞p1 ← point df frmx p1 y + fontheight.
                   p1 y < p2 y ⇒ (dcomp p1 x (df frmwd) p1 y (p2 y - p1 y).)))
    p1 y > p2 y ⇒ ()
    dcomp p1 x (p2 x - p1 x) p2 y fontheight.)!
```

A solution for finding out where you are pointing with the mouse

This routine returns a vector such that

first item	index of character after which you will insert
second item	left x of character
third item	width of character
fourth	top y of character

That is, if we point to character 3, return index 2; point to character 1, return index 0. This will permit forward and backward movement of the cursor in order to select the subset of text. Sending the dispframe the message *mfindc* gives most of the desired information:

☞tv← df mfindc mx my.

tv is now a vector with the correct information with the exception of decreasing *tv[1]* (the index of the character) and accounting for a "feature" of *mfindc*: if you point to the right of last character it tells you the last character--in this case the intention is to append to the end and the returned index should be last, not last-1, and the x position should be mx. The mouse is to the right of the last character if its x position is greater than the character's x position plus the character's width (tv[2] + tv[3]).

```
addto dispframe
  ☞(☟findchar ⇒ (☞t ← mx.
     ☞tv ← SELF mfindc t my.
     tv[1] < 0 ⇒                                  Going outside frame?
       (⇑ {last charx 0 chary})
     (tv[1] = last ⇒ (t > tv[2] + tv[3] ⇒ (tv[2] ← t)
       tv[1]← tv[1] - 1)
     tv[1] ← tv[1]-1.)
     ⇑tv.))!
```

Some other useful additions to basic system classes

☞ bottomscreen ← disp (frmy + frmht)! Where disp is lowest possible window on the screen.

```
addto dispframe                                   Reset frame and window parameters.
  ☞(«dispset ⇒ (☞frmx ← ☞winx ← :.
                ☞winht ← bottomscreen - ☞frmy ← ☞winy ← :.
                ☞frmwd ← ☞winwd ← :.))!

  ☞(«fshow ⇒ ((«of ⇒ (☞buf ← :.                  Reset text information
                ☞last ← :.))
               SELF clear.
               ☞frmht ← 1.
               SELF show))!                       Then show the display.

addto number ☞(«chars ⇒ (⇑ stringof SELF))!
```

Reading the keyboard: Algorithm A

The following routine, as part of the paragraph class definition, will repeatedly handle one character at a time, adjusting *buf* and the index pointers *loc1* and *loc2*. The effect will be to delete, replace, insert, and append to *buf*.

Special characters	Ascii code	
bs	8	
carat	2	(looks like ‸ a small carat character that has 0 width)
del	127	

The following expression assumes we have already computed *loc1* and *loc2*. We want backspacing (bs) to decrease *loc1* and delete (del) to delete the selection (buf[loc1+1 to loc2]).

```
buf[loc1 + 1 to loc2] ← all carat.          Replace each character in the selected text by the 0 width
                                            carat character.
repeat (☞char ← kbd.                        Get character.
        (del = char ⇒ (SELF delete)         Is it delete?
          bs = char ⇒                       Is it the backspace?
        (loc1 > 0 ⇒                         If so, test to see if loc1 is at beginning of text.
          (buf[loc1] ← carat.               If not, can decrease loc1 and replace with the carat.
           ☞loc1 ← loc1 - 1.))              Otherwise, do nothing.

      (loc1 = loc2 ⇒                        Here if character not a backspace. Ordinarily can replace
                                            buf[loc1] by character, and increase loc1; special case exists
                                            if loc1 = loc2. The special algorithm says that a "hole" into
                                            which characters can be stuffed should exist, always
                                            providing extra input space permits replacements larger than
                                            the selected text.
((buf length < ☞last ← last+hole            Start by making certain there is room for the "hole" to
  ⇒ (☞buf ← buf[1 to last]))                be inserted after loc1 (☞hole ← 30).
  buf[loc1 + hole + 1 to last] ←            There's room for the hole so slide over the part to the
    buf[loc1 + 1 to last-hole]              right of loc1 and replace the middle "hole" part with carats.
  buf[loc1+2 to ☞loc2←loc1+hole]←all carat))  The "hole" created has 0 width and therefore is not seen.
  buf[☞loc1 ← loc1 + 1] ← char)             This is always done regardless of input character...
                                            just replace the character.
kbck ⇒ ()                                   See if there is more to do.
pdisp fshow of buf last.                    If not, replace/insert/append/delete completed.
done)
```

Making selection if button pressed: Algorithm B

```
SELF cleanup.                                  Remove old indications of "hole" and of complementing.

SELF showselection of pdisp findchar.          Find the first location and set loc1,loc2 and p1=p2.

repeat

 (button 4 ⇒                                   As long as the button is pressed, keep changing
                                               complemented area and loc2.
  (☞char ← pdisp findchar.                     Get next location.
   char[1] = loc2 ⇒ ( )                        If no change, do nothing.
   ☞t←point char[2] char[4].                   t is new location's point.
   (char[1] < loc2 ⇒ (dfcomp pdisp t p2)       Complement changed area (possibly back to white).
      dfcomp pdisp p2 t)
   ☞loc2 ← char[1].    ☞p2 ← t)                Store the new loc2 and p2.
 done)                                         Done if button 4 not pressed.
☞char ← false.                                 Indicate no characters typed yet.
loc2 < loc1 ⇒ (☞loc2 ← ☞loc1 swap loc2.        When completed, make certain first location
        ☞pt ← ☞p1 swap p2)                     is earlier in the window than second location.
```

Above algorithm assumes the following addition to the class *atom*:

```
addto atom ☞(☟swap ⇒ (☞x←SELF eval.            Lets each instance of atom receive new value
              SELF ← :. ⇑ x))!                 and return the old value.
```

Now the class definition for a paragraph

```
to paragraph t tv |                            temporary variables
      pdisp char buf last loc1 loc2 p1 p2 ht|  instance variables
      dfcomp hole carat bs del                 class variables

(init ⇒ (to dfcomp df p1 p2 (above definition)  Define in context of class.
      ☞hole ← 30. ☞carat ← 2. ☞bs←8. ☞del ← 127)

☟retrieve ⇒ (filin :.)                         Create instance from a file; filin checks if value is a file.

☟ store ⇒ (☞t ← file :.                        Write text such that, when evaluated
      t ← 'sched ← paragraph of '              creates instance of a paragraph
      t ← 39. t ← buf. t ← 39.                 and stores in scheduler.
      t ← ' '+last chars + ' at '+ pdisp (frmx chars + ' ' +
            frmy chars + ' ' + frmwd chars).
      t close.)
```

CREATE INSTANCE

```
isnew ⇒ (☞pdisp ← dispframe 0 1 0 fontheight string 0 noframe.

                                               Create display area.
      ☞buf ← (☟of ⇒ (:) string 0)
      ☞loc1 ← ☞loc2 ← ☞last ← buf length.      Create indices.
      ☟at ⇒ (SELF show at (:) (:) :)           If told where, show area.
      pdisp frame black.)                      Frame the window.
```

SEE TEXT

```
show ⇒ ((at ⇒ (pdisp dispset (:) (:) :))      Reset display location.
        pdisp fshow of buf last)               Tell text information.
```

SCHEDULING MESSAGES

```
startif ⇒ (⇑pdisp hasmouse)                    Condition for starting is mouse inside the area.

quitif ⇒ (⇑pdisp hasmouse is false)             Quit if mouse no longer in area.

firsttime ⇒ (pdisp hasmouse ⇒ (↑ht ← pdisp frmht. SELF showselection))

eachtime ⇒ (kbck ⇒                             Typing anything?
             (Algorithm A)                     Keyboard algorithm
           button 4 ⇒                          Pressing button to make new selection?
             (Algorithm B))

lasttime ⇒ (pdisp's (↑frmht ← ht).             Reset frame height to clear black frame.
           pdisp frame white.
           SELF cleanup.
           pdisp frame black.)
```

MANIPULATING THE TEXT

```
showselection ⇒                                Upon entering the window, set the cursor at the end for
                                               automatic append; can receive parameter values from the
                                               message.
    (↑tv ←                                     Determine value of first selection: as message or as last
     (of ⇒ (:)                                 text character.
      {last  pdisp (last=0 ⇒(frmx)charx) 0 pdisp (last= 0 ⇒ (frmy)chary)}).
      ↑loc1←↑loc2←tv[1].
      ↑p1 ← point tv[2] tv[4].
      ↑p2 ← point tv[2]+1 tv[4].               p2 is a little wider to help "see" current place.
       dfcomp pdisp p1 p2.)

delete ⇒                                       Reorganize buf removing text between loc1, loc2.
  (↑buf ← buf[1 to loc1]+buf[loc2 to last].
       ↑loc2 ← loc1. ↑p2←point p1 x+1 p1 y.
       ↑last ← buf length)

cleanup ⇒ ((char ⇒ (↑char ← false.             Clear the window of complemented text and
       ↑buf ← buf[1 to loc1] + buf[loc2 + 1 to last].      remove the "hole".
       ↑last ← buf length.
       ↑loc2 ← loc1.))
      pdisp fshow of buf last)

is ⇒ (ISIT eval)

's ⇒ (⇑ 8 eval))!

paragraph init!
```

It is possible to schedule a paragraph text editor by typing

sched ← paragraph!

window appears in the upper left corner of the screen with no width or height.

sched ← paragraph at 100 50 200!

window appears at 100, 50 with width of 200.

sched ← paragraph of 'I am a text editing window'!

window appears in upper left corner with the text showing.

Classes for Building Models

"Simpula": Simula-style Simulation

We have chosen a simple example of a scheduling mechanism for building simulations of dynamic environments such as hospitals and classrooms. The basis for this example is the simulation language Simula (a major inspiration for Smalltalk).

The basic entities of Simula are instances of classes and ALGOL-like data-types. Simula simulation operates primarily through scheduling pseudoparallel processes by means of a sequencing set which holds the quiescent processes sorted by desired time of activation. Associated with each process are the object itself, the time the object is scheduled to wakeup and do something, and a message telling the object what state to go to next. This message was either constructed by the object when it last ran, or is a default message (we will use *run*).

There is a system time (*now*) which indicates where the simulation's progress has currently reached. All activation times in the sequencing set are equal to or greater than the system time. A great idea of Simula is that system time is not advanced until there is no more computing to be done by the currently active event. This means that an event can consume an arbitrary amount of computing power; then, if there is nothing scheduled for the next one hundred (simulated) years, the system time will be advanced one hundred years without any "clock ticking" in between.

An item in the sequencing set (*SQS*) is an instance of an *Event Notice*, a simple structure containing the object to be activated (*ob*), the desired event time (*etime*, a floating point number), the message telling the object what state to go to next (*msg*), and *next* and *prev*--indicators to the next and previous elements in the sorted set.

Event Notice

ob	msg	etime	prev	next

SQS

Event Notice	Event Notice	Event Notice

Note that one object can be scheduled as more than one event, each event applying a different message to (requesting a different activity from) the object. Hence we place the message in the Event Notice rather than storing it as information local to the object. This is an improvement over Simula which only allows one phase of an event to be scheduled. The main activity of the SQS will be to add to, delete from, and sort the set of Event Notices. This job differs according to where the event time is stored, that is, in the Event Notice or more local to the object.

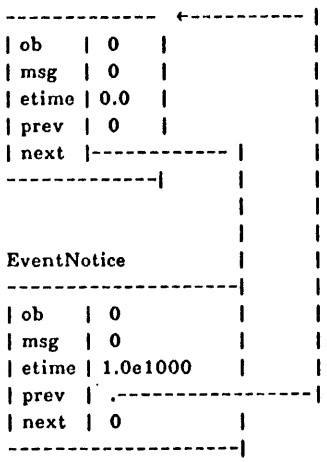
```
to EventNotice prop | ob msg etime prev next

(isnew ⇒ ( ☞ob←:. ☞msg←:. ☞etime←:. ☞prev←:. ☞next←:.)

◀'s⇒ (☞prop ← ▯. ◀← ⇒ (⇑prop ← :) ⇑prop eval)

◀is ⇒ (ISIT eval))!
```

The sequencing set is not much more complex. It maintains *now*; *current*, the current process under activation; and the ordered set of Event Notices, *set*. In order to make this explanation simpler, we will include two dummy Event Notices with event times 0.0 and "infinity", that will, by default, be the first and the last elements in the set. This means that we will not have to check for a special termination condition, and that we avoid the possibility of a circular list. We always select the second item in *set* as the next active event.

We need to provide messages to *schedule* a new event, to *remove* an event, and to *activate* the next event, as well as initialization for the set itself.

Simpula ← SQS!

Initializing the set means to create two Event Notices, the first and last scheduled events. The event objects are meaningless, so we choose to define them as 0; the first time is 0.0, which is also the value of now; the last is a large number like 1.0e1000. Simpula's set is an Event Notice linked in an ordered chain to other Event Notices



Simpula schedule blob!

An EventNotice, whose object is blob, whatever it may be, is added to the chain of events. By default, its msg is (run) and its event time is the same as now. In the above example, this new EventNotice will be the second event. The (default) event whose object is 0 and event time is 1.0e1000 is always the last event in the set.

Simpula schedule blob for (changeplace) at 20!

The object is scheduled as explained above, but the msg is (changeplace) and the event time is Simpula's now+20.

Simpula activate!

Get the next scheduled event (newOb), set current to newOb's object, set now to newOb's etime, and send current newOb's message.

Simpula remove!

Takes and returns the next event off the set, meanwhile reorganizing the chain of Event Notices.

Simpula full!

Reply is true if there is an event, other than the two dummy events, scheduled.

```
to SQS  finger newOb time msg | now current set

    (isnew ⇒ (☞set ← EventNotice 0 0 0.0 0 0.
              set's next ← EventNotice 0 0 1.0e1000 set 0.
              ☞now ← 0.0)

     ◀schedule ⇒ (☞newOb ← :.
                  ☞msg ← (◀for⇒(:) ☞(run)).
                  ☞time ← (◀at ⇒ (now + :) now).
                  ☞finger ← set's next.
                  repeat (time ≥ finger's etime ⇒ (☞finger ← finger's next. again)
                  ☞newOb ← EventNotice newOb msg time  finger's prev finger.
                  newOb's prev's next ← newOb.
                  finger's prev ← newOb.
                  done))

     ◀activate ⇒ (☞newOb ← SELF remove.
                  ☞now ← newOb's etime.
                  ☞current ← newOb's ob.
                  apply current to newOb's msg)

     ◀remove ⇒ (☞finger ← set's next.
                finger's next's prev ← finger's prev.
                finger's prev's next ← finger's next.
                ⇑ finger)
     ◀full ⇒ (⇑ 0 ≠ set's next ob)

     ◀print ⇒ (☞finger ← set's next.
               repeat (0 = finger's ob ⇒ (done)
                    finger's ob print.
                    ☞finger ← finger's next.))

     ◀'s ⇒ (⇑ 8 eval)
     ◀is ⇒ (ISIT eval))!
```

The above definitions are quite general, having two properties that might not be necessary in some applications: (1) backwards pointers for an EventNotice which take extra time to rechain, and (2) the instance variable *current* for SQS. As a response to the message *activate*, we say

```
☞current ← newOb's ob.   apply current to newOb's msg.
```

We might instead have

```
evapply newOb's ob to newOb's msg.
```

eliminating the instance variable.

Now to test it out.

By convention, a scheduled object, such as a *blob*, must respond to the default message *(run)* or to some equally useful activation message.

```
to blob x y | sides 😇
  (isnew ⇒ (☞sides ← 0. ☞ 😇 ← turtle. 😇 width ← 2.  SELF draw)
    ◀run ⇒ (SELF undraw.  SELF draw.
            Simpula schedule SELF at avgwaitime+rand mod 100)
    ◀draw ⇒ (😇 penup goto ☞x ← rand mod 500 ☞y ← rand mod 500 pendn up.
            poly ☞sides← (sides + 1) mod 7.
            😇 penup goto x y pendn up. )
    ◀undraw ⇒ (😇 white. poly sides. 😇 black)
    ◀print ⇒ ( )
    ◀'s ⇒ (⇑ 8 eval)
    ◀is ⇒ (ISIT eval))!

to poly  s
    (do (☞s←:) (😇 go 10 turn 360/s))!

☞i ← 13!
☞avgwaitime ← 100!

to rand  (⇑ ☞i ← i * 5)!
```

Try

```
☞Simpula ← SQS!

Simpula schedule blob!

Simpula schedule blob!

repeat (Simpula activate)!
```

or

```
PUT USER ☞DO ☞(kbck ⇒ (ev) Simpula activate)!
```

The result is two polygons bouncing around the screen. With the modified USER task, it is possible to temporarily interrupt the bouncing in order to type some messages (such as scheduling another blob or examining the scheduled events).

Note, another *rand* expression, that avoid the need to initialize the variable and also allows ranges to be specified, is given below.

```
rand!

rand between 10 40!

to rand low high | | n
(☞n ← (null n-> (13) n*5).
  ◀between ⇒ (☞low ← :. ☞high ← :.
          ⇑ low + n mod high + 1 - low)
  ⇑ n)!
```

A Simple Hospital Simulation

A *hospital* will be composed of *departments* (including admissions, surgery, wards, labs), each of which has a number of *resources* (such as attendants, doctors, beds, operating tables) and *patients*. A typical *patient* (there will be many of them) has a *name, age*, and so on, a *schedule* which contains a route through the hospital specified at admissions, and a reference to the patient's current department. The patient visits the indicated department on the schedule, stopping at the department's front desk to check to see if there is a resource available for him. If there is, the patient will occupy that resource for some average *treatment time*. If there is no resource available, the patient must wait indefinitely on the *department's line* until one is available. After consuming the resource, the patient will check the waiting line and send the next waiting patient, if any, to the desk.

If this simulation is set up with typical entrance intervals and treatment times found in a given hospital, an examination of the department's lines after the simulation is in progress will give some insight into the "bottleneck" departments of the hospital.

The hospital can include a Smalltalk vector of elements, each of which is a department.

```
☞dept ← vector 20!
```

will contain 20 departments. A department has two main parts: resources available and its waiting line. It also has a name and an average treatment time for each patient.

```
to department  prop | resources line available nme treatime

    (isnew ⇒ (☞available ← ☞resources ← :. ☞line ← SQS.
              ☞nme ← stringof 8.
              ☞treatime ← (☟time ⇒ (:) avgwaitime))

    ☟take ⇒ (☞available ← available - 1)

    ☟giveup ⇒ (☞available ← resources min available+1)

    ☟'s ⇒ (☞prop ← 8. ☟← ⇒ (⇑ prop ← :) ⇑ prop eval)

    ☟is ⇒ (ISIT eval))!
```

Initialize the departments for 1 to 5 resources.

```
for j to dept length (dept[j] ←
        department rand between 1 4 noname time 20* rand between 0 4)!
```

We have to define a typical patient.

```
☞routine ← stream of {dept[3] dept[6] dept[7]}!
```
Setup for the patient's schedule.

```
routine reset.!
```
Reset the stream to the first item.

```
☞jane ← patient 'jane' 22 routine!
```
Create jane as a patient whose name is jane, age 22, schedule to be in three departments: 3, 6, and 7. Notice that the third message to patient must be an instance of stream

```
Simpula schedule jane for ☞(wakeup)!
```
The patient is scheduled to wakeup now.

```
jane wakeup!
```
The patient schedules herself to visit the department (newplace).

jane visit!

The patient sees if there are available resources in the department. If so, the patient takes a resource and schedules herself to move on after the department's treatment time; otherwise, she enters the waiting line.

The patient can also receive this message by being removed from the department's line and scheduled again for a visit.

jane treatment!

The patient gives up her resource in the current department and wakes up the next patient, if any, in the waiting line; if there are other departments to visit, she schedules herself to visit the next one.

```
to patient prop | nme age schedule newplace

    (isnew ⇒ (☞nme ← :. ☞age ← :. ☞schedule ← :.
              ☞newplace ← schedule next)

    ☞visit ⇒ (newplace's available > 0 ⇒
              (newplace take.
               Simpula schedule SELF for ☞(treatment) at newplace's treatime)
               newplace's line schedule SELF for ☞(waiting).)

    ☞treatment ⇒ (newplace giveup.
                  (newplace's line full ⇒ (☞prop ← newplace's line remove.
                                           prop's ob wakeup))
                  schedule end ⇒ ( ) ☞newplace ← schedule next. SELF wakeup.)

    ☞wakeup ⇒ (Simpula schedule SELF for ☞(visit))

    ☞print ⇒ (nme print. sp.)

    ☞'s ⇒ (☞prop ← 8. ☞← ⇒ (⇑prop ← :) ⇑prop eval)

    ☞is ⇒ (ISIT eval))!
```

All that remains is to make up an admittance process which creates new patients at reasonable intervals. We can add mechanisms for stopping the simulation and asking departments about their resources and waiting line as well as asking patients about their schedule. A patient might also know his disease and keep around a history of waiting times. A query method can be implemented by creating a display window (a talkwindow) that can be scheduled in Simpula. Any time a key is pressed on the keyboard, the window is scheduled to wakeup and expect inquiries about objects in the environment. The main USER task might be

```
kbck ⇒ (Simpula schedule talkwindow for ☞(wakeup))
Simpula full ⇒ (Simpula activate)
```

Graphic feedback can be provided by having a department print itself as a rectangular area containing marks for each resource. The marks can be differently colored depending on whether or not they are available. The department might have three stations: a desk for the patient coming to *visit*, a waiting room for the *waiting* patient, and a staff room for the patient under *treatment*. We can also add a graphic representation of the system clock to display the value of *now*. Pressing a mouse button can indicate that you are making an enquiry about a particular department or patient. The department or patient has a graphic representation that is pointed at by the mouse cursor. The intention of pointing at the object is to schedule it for talking about itself. The USER task might now be

```
kbck ⇒ (Simpula schedule talkwindow for ☞(wakeup))
0 < mouse 7 ⇒ (Simpula schedule (findobject at mx my) for ☞(talk))
Simpula full ⇒ (Simpula activate)
```

Instances of patient and department should respond to the message *talk*.

INDEX

This index was prepared from a Smalltalk Information Storage and Retrieval System in which the contents of the sections of the manual are referenced. As a result, the pages given below generally refer to the beginnings of the sections in which the information can be found. We have identified three types of indexed items: basic system classes, utilities, and examples created especially for this manual. The basic classes and utilities are provided in the Smalltalk system when you type *resume small.sv*. The index distinguishes between pages where the items are defined (def) and those where the item is referenced (ref).