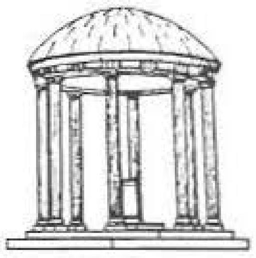
Programming in Smalltalk

TR87-023

April 15, 1987

Katherine N. Clapp, Hermann von Borries

The University of North Carolina at Chapel Hill
Department of Computer Science
Sitterson Hall, 083A
Chapel Hill, NC 27514



Programming in Smalltalk

Katherine N. Clapp Hermann von Borries

April 15, 1987

Contents

Chapter 1

Introduction

1.1 Motivation

The Smalltalk system is a remarkably powerful programming environment. Complex applications can be designed and coded in Smalltalk far more easily and quickly than in most programming languages. Three or four lines of Smalltalk code can have the same power as pages of C code.

Smalltalk is an object-oriented programming language. Programming consists of defining objects and making them perform actions. For example, you can display a spiral on the screen by creating a pen and sending the message drawSpiral to this pen.

Examples of objects are numbers, arrays, text, rectangles, a terminal screen, an editor, and a compiler. Useful actions might include move: aDistance (defined on rectangles and text), insert: aText (defined on editors), and divideBy: anInteger, (defined on numbers).

The object-oriented programming paradigm has several advantages over traditional programming languages such as C or Pascal. It enforces modular design and implementation, where modules correspond to objects and their methods. It encourages programmers to build libraries of tools and to program incrementally, building on those tools. The power of Smalltalk comes from the tools (objects and methods) it provides. Smalltalk programmers customize, extend, and piece together existing tools to create their programs.

Unlike most common programming languages, Smalltalk is a comprehensive environment which includes its own text editor, compiler, and sophisticated debugging facilities. The environment is fairly easy to learn. You can manage most components with the small set of commands presented in this manual.

Smalltalk also contains a large amount of source code you will want to reuse. In fact, Smalltalk is itself written in Smalltalk, and all the source code is readily available. Most of the modules of Smalltalk are very general; you will find many of them useful, although sometimes not well documented.

1.2 Overview

This manual is a practical guide to Smalltalk for those who want to design and program simple applications quickly. With this manual and the documentation you will find in the Smalltalk code, you should be able to master a useful subset of the system. Sources of more detailed information are listed in the Reference section.

The best way to read this guide is sitting at a Sun running Smalltalk. You can read the chapters in any order you wish, though we recommend reading them sequentially. The guide contains the following chapters:

Chapter 1: Introduction

An overview of this manual.

Chapter 2: Tutorial

Steps through a sample session with Smalltalk, for those who have never used the system. Presents the basic building blocks of the Smalltalk system: objects and methods. Covers essential functions such as startup, quitting, and saving.

Chapter 3: Classes

Introduces the hierarchy of classes available in the system. Discusses in some detail a few objects which are likely to be useful.

Chapter 4: Programming

Describes programming in Smalltalk. The syntax and semantics of the Smalltalk language are discussed; you will learn how to define classes and write methods.

Appendix A: Reference

Points to sources of more information: books and articles.

Appendix B: Reading Smalltalk Source Code

A guide to reading source code.

Appendix C: Programming the Interface

Describes how to program tools such as the mouse and pop-up menus.

Appendix D: Sample Program

Lists the code for the example used in the Programming chapter.

1.3 Definitions

As you may have noticed, Smalltalk includes its own vocabulary. This section defines basic Smalltalk concepts.

Method

A method is a procedure or function. Methods are code segments, they can be called, receive parameters, and return a value.

An example of a method is drawSpiral, defined on pens.

Sending a message

To "send a message" is to call a method. The message consists of the name of the method and the parameters of the call. The method can return a value as an answer.

(Don't get confused: method calls are not asynchronous. Smalltalk's message sending has nothing to do with message passing.)

For example, the message goto: location can be sent to a pen. The name of the method is goto: and the parameter is location.

Class

A class is a definition which includes:

- The definition of a data structure.
- Methods to operate on the data structure.

Classes correspond to abstract data types: an abstract data type consists of a data structure and operations (Smalltalk methods) to operate on that data structure.

Programming in Smalltalk consists of writing classes.

An example of a class is Pen. The data structure includes a description of the current location of the pen, whether it is up or down, its color, its shape, etc.

Object

An object is an instance of a class.

At runtime, a program can request a new object from a class. This operation will:

- Set aside memory for the data structure.
- Associate this new object with the methods of its class.

In other words, a class is a *template* for an object: new objects are created according to the definition of the data structure and the methods of the class.

Objects have a flavor of autonomy. Once you create one, you lose the direct control over what goes on inside the object and the only way to do something with or to the object is to send it a message. This enforces information hiding.

Almost everything in Smalltalk is an object: numbers, arrays, sets; even methods and classes.

Examples of objects are different pens: a black pen, a big pen, etc.

Variable

The Smalltalk notion of variables differs in two ways from other programming languages:

All variables in Smalltalk *point* to objects; they do not contain values like in other programming languages.

Variables are not typed (but the objects are!). This means that a variable is only named, not declared. Any variable may point to an object of any class.

Instance variable

Instance variables are the set of variables which contain the data structure of an object.

The instance variables are defined in the class. Whenever a new object is created, it will have the number of instance variables specified in the class definition.

Instance variables are owned by the object; they are *not accessible* except through methods belonging to the object's class.

For example, a pen has instance variables color, location, penDown, etc.

The following table compares the Smalltalk terms to terms in other programming languages. The comparison is not formally exact, but it provides a starting point to understand the new "lingo":

Smalltalk-80	Pascal or C
method	procedure or function
class	abstract data type (module, Ada package)
object	strongly typed variable
message	procedure call
send a message	call a procedure
variable	pointer to a data structure

Table 1.1: Informal comparison of Smalltalk terms

This a diagram describes the relationship between objects, classes, methods and instance variables.

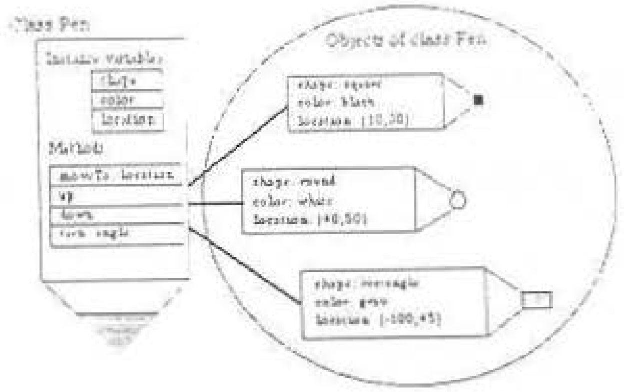


Figure 1.1: Example of objects, classes, methods and instance variables

Chapter 2

Tutorial

2.1 Overview

This tutorial introduces the novice Smalltalk user to the system. It defines basic terms needed to understand other chapters of the manual. If you have never used Smalltalk, be sure to go through this exercise.

At UNC, Smalltalk is available on Sun Workstations. Because it uses a lot of memory, Smalltalk is significantly faster on 4-megabyte Suns than on 2-megabyte machines. There is also a version of Smalltalk for the MacIntosh, but this tutorial assumes that you are using a Sun.

▷ This symbol indicates that you should perform an action at the terminal.

2.2 Getting Started

2.2.1 Setup Procedure

▷ Create and enter a directory from which to run Smalltalk:

```
mkdir newDirectory
cd newDirectory
```

▷ Execute the UNC setup procedure

```
/unc/smaltalk/ps0/bin/setup
```

The single l in smaltalk is not a misprint! All six files created by the setup procedure are needed to run Smalltalk. Always start the system from this directory.

2.2.2 Startup

▷ To start Smalltalk, type

```
st
```

The initial Smalltalk screen looks like figure 2.1. It may vary from installation to installation.

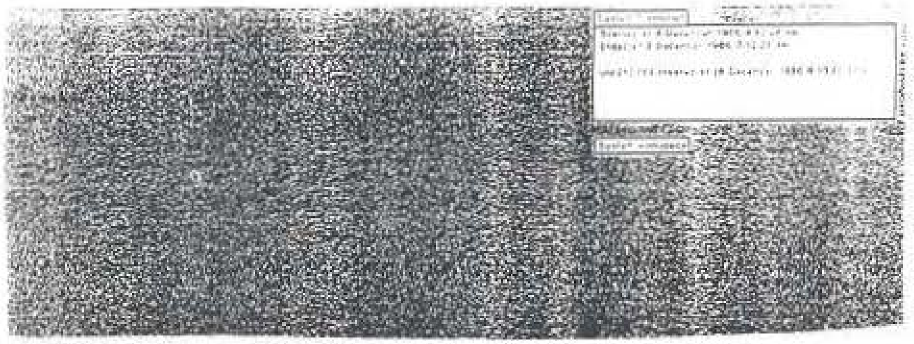


Figure 2.1: Initial Smalltalk screen (installation dependent)

(Smalltalk does not run properly within suntools. If your login file executes suntools automatically you will need to change it.)

2.2.3 The Mouse

The Smalltalk mouse has three buttons. In the literature and in Smalltalk code you may see references to red, blue, and yellow buttons. Since modern mice are monochromatic, this document refers to left button (red), middle button (yellow), and right button (blue).

Each one has a designated use:

left button

Press this button to *select* text and to position the cursor, when the cursor is in a window. It has no effect if the cursor is not in a window.

middle button

accesses a menu for using the *contents* of a window in various ways. The menu varies from window to window.

right button

accesses a menu to *modify* a window: open, close, change size, and others.

2.2.4 Menus

Most work in Smalltalk is done with menus. This is a typical menu:

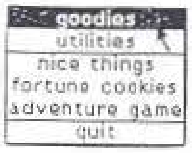


Figure 2.2: A menu

Smalltalk displays a menu when you press a mouse button. The current option is highlighted. You select an option by releasing the mouse button.

If you don't want to make any selection, move the cursor off the menu and then release the menu button. If a menu flashes on the screen, you must select one of its items before going on to other work.

2.2.5 Using The Main Menu

▷ Press the middle button while the cursor is outside all windows.

The *main menu* appears:

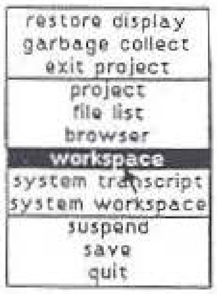


Figure 2.3: The main menu

Exercise this menu:

▷ Select the option restore display and release the middle button.

Smalltalk will erase the screen and redraw all windows.

▷ Get the menu again, and move the cursor off the menu. Release the middle button.

The menu disappears and no action occurs.

2.3 Windows

2.3.1 Creating a Window

This section demonstrates how to create a window: a *workspace*. A workspace is a general-purpose type of window. You can put any kind of text in a workspace. You can also use one to code small programs (see chapter 4).

▷ Select the workspace option from the main menu:

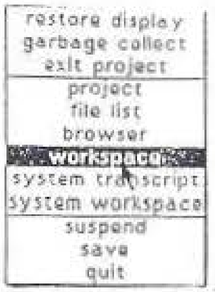


Figure 2.4: Main menu option to create a workspace

Now the cursor will transform into the top left corner of a window:

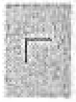


Figure 2.5: Top left cursor

You can move this corner freely around the screen by moving the mouse button.

▷ Press and hold the left mouse button to fix the location of the new window's top left corner. Keep holding that button!

Now the cursor will look like a bottom right corner. A rectangular form will flash on the screen in the space delimited by the corners. As you move the mouse, the form will expand or contract.

▷ Release the button to fix the bottom right corner of the window.

You have just opened a workspace:

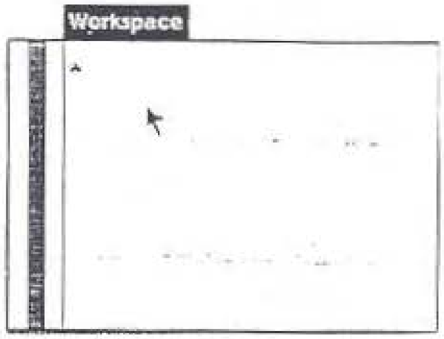


Figure 2.6: A workspace

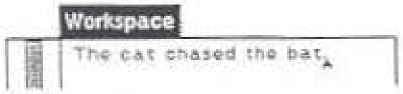
2.3.2 The Editor

The Smalltalk editor is available in the workspace and all windows where you can enter text. It is simple and well-adapted to Smalltalk programming. It is not possible to use other editors inside the system.

▷ With the cursor inside the workspace, click the left mouse button.

The *edit cursor*, a caret, appears. To change the cursor location, press the left button at a new spot in the window.

▷ Type *The cat chased the bat.*

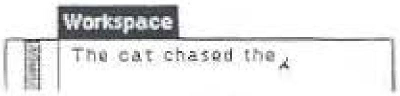


The cat chased the bat.

Figure 2.7: Typing text

Text is inserted at the cursor location.

▷ Place the cursor at the end of *bat* and hit backspace three times.

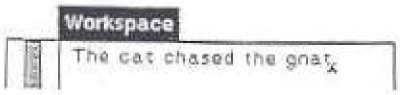


The cat chased the

Figure 2.8: Deleting text

This is one way to delete text. More efficient ways are explained below.

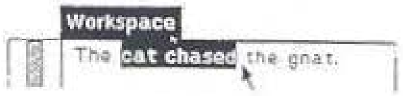
▷ Type *gnat*.



The cat chased the gnat.

Figure 2.9: Typing new text

▷ Now select the words *cat chased*: place the cursor at the beginning of *cat*. While holding the left button down, move the cursor to the other end of *chased*. Release.

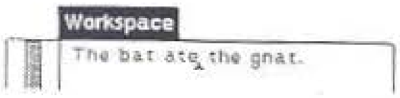


The cat chased the gnat.

Figure 2.10: Selecting text

The selected text is highlighted. You could de-select it by clicking the left button once. Or you could replace the text by typing.

▷ Type *bat ate*.



The bat ate the gnat.

Figure 2.11: Replacing text

You can select various units of text with the left mouse button as follows:

a word

double-click at the end of the word.

delimited text (inside quotes, parentheses or brackets)

double-click at either end of the passage, *after* the first or *before* the last delimiter.

all text

double-click at the beginning or end of the text.

a line

double-click at the end of the line.

Erase selected text by pressing the delete key.

There is a menu, accessed by the middle button in editable sections of windows, which contains other edit commands:

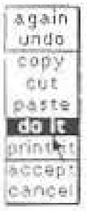


Figure 2.12: The Edit Menu

undo

reverses the last edit command (usually!).

copy

copies any text currently highlighted into a buffer.

cut

deletes highlighted text and saves it in a buffer.

paste

adds the text currently in the buffer at the location of the cursor. If any text is currently selected (highlighted), it is replaced.

accept

reads the text, and stores it internally in Smalltalk.

If the window accepts source code (like the workspace or the browser), it is also checked for syntax errors. If there is an error, an explanatory message appears in the window. You have to delete the message (press delete), correct the error, and accept again.

cancel

reverts to the text as it was before the last accept.

Chapter 4 covers the options do it, print it, and other uses of accept. Refer to [2] for a complete description of the editor.

2.3.3 Window Manipulation

You manipulate windows using the right button menu:

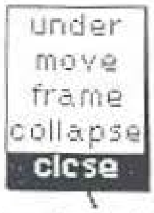


Figure 2.13: Standard right button menu

collapse

This leaves only the window's label visible on the screen. The top left corner cursor allows you to move the label to some place on the screen.

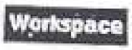


Figure 2.14: Collapsed window

frame

lets you resize the window.

move

allows you to move the window, without changing the size.

close

removes this window permanently.

▷ Test the menu options on your workspace window.

▷ With the cursor in the workspace, select close from the right button menu. Since there is unsaved text in the window, Smalltalk will present a confirmer menu:

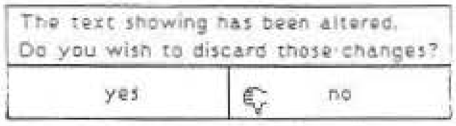

The text showing has been altered.
Do you wish to discard those changes?

Figure 2.15: Confirmation for closing a window with unsaved text

▷ Move the cursor to the square marked yes and click the left mouse button.

2.3.4 Scroll Bars

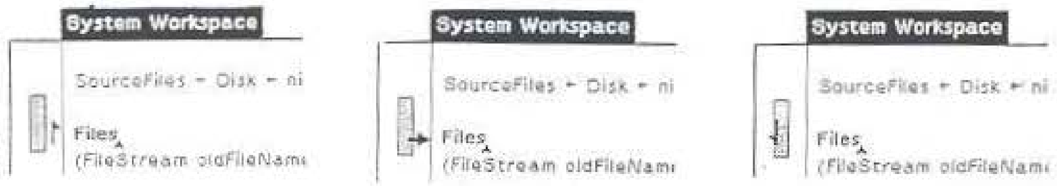
Most windows or subwindows have a *scroll bar* along the left edge. A scroll bar is visible only if the cursor is in the window. The shaded part of a scroll bar indicates the portion of the text currently visible in the window.

Now exercise the scroll bar in a system workspace.

▷ Open a system workspace. If a label is already present on your screen, use the frame option of the right button menu to frame the window. Otherwise, use the main menu to create a new one.

▷ Place the cursor in the system workspace.

▷ Move the cursor into the scroll bar which appears along the left edge of the subwindow.



```
SourceFiles ← Disk ← ni
Files
(FileStream oldFileNam
```

```
SourceFiles ← Disk ← ni
Files
(FileStream oldFileNam
```

```
SourceFiles ← Disk ← ni
Files
(FileStream oldFileNam
```

Figure 2.16: The cursor in the scroll bar

The cursor becomes an arrow. As you move the cursor around inside the scroll bar, the direction of the arrow will change. Click the left button when the arrow points *up* to page *down* one line; click when it points *down* to page *up* one line. If the cursor is inside the vertical stripe in the scroll bar, it appears as an arrow pointing right.

You can jump several pages when the cursor is a right arrow: the position of the cursor in the scroll bar determines the amount scrolled. You can also hold the left button down while moving the cursor along the scroll bar to move around in the contents of the window.

2.3.5 Types of Windows

Smalltalk provides special-purpose windows to help you find and structure information as well as write new code. Most types of windows can be created by selecting their names from the main menu:

browser

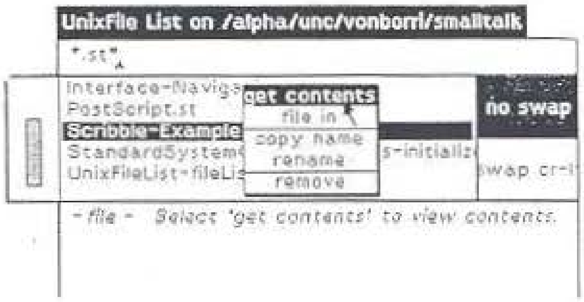
A browser window is the main programming tool. It is also extremely useful for learning about the objects which make up the Smalltalk system. Section 2.3.6 describes the browser, and Chapter 4 explains how to program with it.

workspace

A workspace is a blank window to be used any way. You can write short programs in this window. The workspace is especially useful for writing code which is not going to be saved.

file list

At UNC, the file list window lets you read and edit Unix files while you are in Smalltalk.



```
UnixFile List on /alpha/unc/vonborri/smalltalk
*.st*
Interface-Naviga
PostScript.st
Scribble-Example
StandardSystem          s-initializ
UnixFileList-fileLis
- file -  Select 'get contents' to view contents.
```

Figure 2.17: A file list window

In the top portion of the window, enter the file(s) you want listed. You can use regular expressions here. For example, to list all Smalltalk code files (which always end with .st), type *.st. Use accept on the middle button menu when you are ready.

In the next window you will see list of files which match your description. If you select a file and then press the middle button and get contents, the file will be listed in the bottom window.

Use the bottom window to edit the file if you wish. When you finish, select put from the yellow button menu to write the file back on disk.

system workspace

The system workspace contains documentation and templates to help you issue commands to manipulate files, do error recovery, and find out information about methods.

This window is normally present when you load Smalltalk, so you don't need to create it.

system transcript

This window provides a record of important occurences in your Smalltalk sessions. It also displays informative error messages.

This window is also normally present, so you don't need to create it.

For an explanation of the other main menu options, see [2].

There are many other types of windows, like the inspector (see Chapter 4) or the debugger window, which appears whenever there is a runtime error.

2.3.6 The Browser

▷ Select the browser option of the main menu and frame the window.

You have just opened a browser. Browsers are the most important type of Smalltalk windows. You use them to examine and modify classes and their associated methods.

Classes and methods are organized into *categories* for ease of use. Inspect your browser. The four sub-windows along the top contain, from left to right:

- categories of classes
- classes
- groups of methods, known as *protocols*
- methods.

If you select a category in the first window, classes contained in that category will appear in the next sub-window. If you select a class, method protocols for that class will appear in the next window. And if you select a protocol, a set of methods will be displayed in the fourth window. Finally, Smalltalk code for a selected method will appear in the large bottom window.

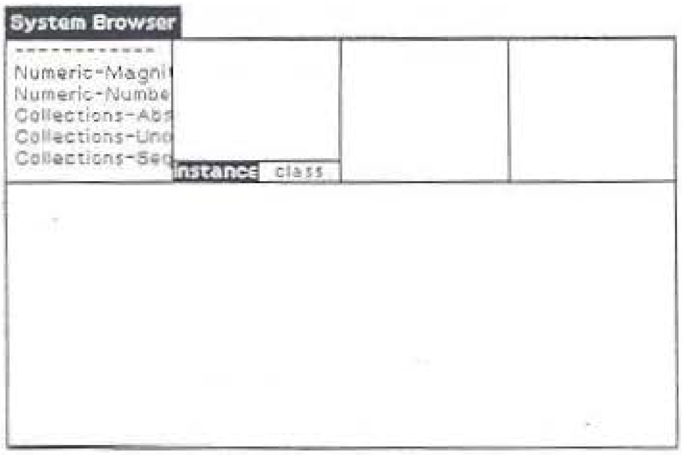


Figure 2.18: The Browser

▷ Move the cursor into the top left subwindow, and scroll the contents of the window up until you see the category Graphics-Primitives.

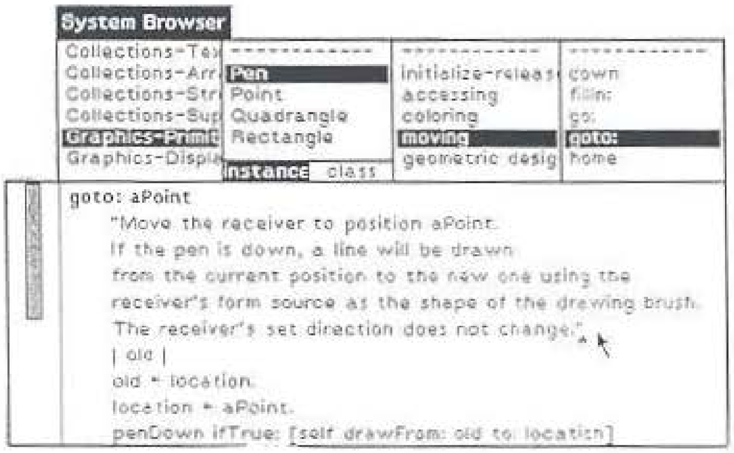
▷ Select Graphics-Primitives from the list in the top left subwindow.

All classes in this category are now displayed in the window to the right.

▷ Select Pen from the second window.

When Pen is selected, you can see the types of methods provided for this class displayed in the third window.

⊳ Now select a protocol from the third window and a method from the fourth window. You will see the source code of the method in the bottom subwindow:



```
goto: aPoint
    "Move the receiver to position aPoint.
    If the pen is down, a line will be drawn
    from the current position to the new one using the
    receiver's form source as the shape of the drawing brush.
    The receiver's set direction does not change."
    | old |
    old ← location.
    location ← aPoint.
    penDown ifTrue: [self drawFrom: old to: location]
```

Figure 2.19: A method of class Pen in the browser

⊳ Browse through some of the methods for Pen by selecting first a protocol and then a method.

2.4 Executing Examples

2.4.1 Execute a method for Pen

There are two types of methods: *instance* methods and *class* methods. Instance methods cause a specific object (an instance of a class) to do something. If you create a particular pen that is black and is named *bic*, you use instance methods to make it draw or move or change color. Some classes include examples as class methods to demonstrate potential applications.

⊳ Select the class box in the browser.

New protocols appear in the third window. These are groups of *class methods*.

▷ Select examples in the third window, and then select example in the fourth window.

In the bottom window you will see the code for this example using a pen. Execute the example as follows:

▷ Select the words Pen example (without the quotes) written in a comment at the end of the code. Refer back to section 2.3.2 if you are unsure how to select text.

The words Pen example are written inside comments simply for convenience; you could also type Pen example yourself and then select that text.

▷ Now send the message. With the cursor anywhere in the code window, choose do it from the middle button menu.

The cursor will have an asterisk next to it until execution completes. Then, the scroll bar will reappear in the subwindow where the cursor is. The result should be a rather impressive spiral!

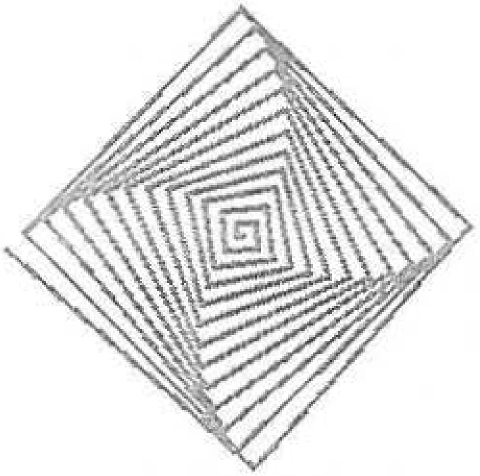


Figure 2.20: The spiral

2.4.2 Modify a Method

▷ Now draw the spiral in white instead of gray: replace the line

```
bic mask: Form gray.
```

with

```
bic mask: Form white.
```

The easiest way to do this is to select *gray* and type *white*.

▷ To draw a thicker spiral, change the width of the drawing pen:

```
bic defaultNib: 4.
```

becomes

```
bic defaultNib: 8.
```

▷ Inform Smalltalk of your changes by pressing the middle mouse button and selecting accept from the menu which pops up. Then execute the example again.

2.4.3 Display Text on the Screen

▷ Select Graphics-Display Objects, just under Graphics-Primitives in the top left window of the browser. Then select the class DisplayText. Make sure that the class box is highlighted (not the instance box) and select the method protocol examples from the third window. Finally, select example in the fourth box.

▷ Test this method as described above: select and execute the words DisplayText example.

As you move the cursor, text will appear on the screen. Press any button to terminate. Try altering the method by replacing the string in single quotes with any text. Accept the changes and re-execute the method.

▷ Select restore display from the main menu to clear the text off the screen.

▷ The Smalltalk system has many other interesting examples; explore and try some! In particular, you might want to look at examples for the class Form (in the Graphics-Display Objects category), Circle and Arc (both in the Graphics-Paths category).

2.5 Saving your Work

2.5.1 File Out

The most economical way to save your code is to *file out* a category, class, protocol, or method; then you can file it in in another session (see below). For example, if you have changed several of the methods for Pen, you can save the entire class Pen as follows:

▷ Select the class Pen in the second window of the browser.

▷ With the cursor in the second window, select file out from the middle button menu.

▷ Wait until the cursor is a up-left arrow again.

You could save a single method by selecting one in the fourth window and choosing file out from the menu in the fourth window. You could save an entire class category selecting a category and filing out with the cursor in the first window. A new file, xxx.st, will appear in your current directory, where xxx is the name of the category, class, protocol, or method.

2.5.2 File In

You can *file in* a file that you have previously filed out with the file in option in the File List window:

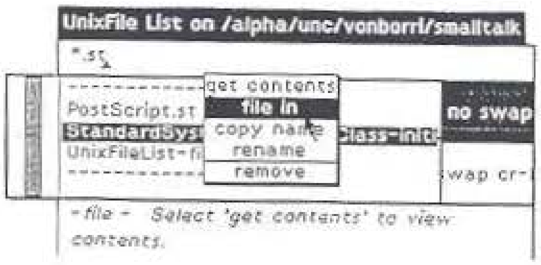


Figure 2.21: Filing in with the File List

You also could find the Smalltalk statement (FileStream oldFileNamed: 'my-file.st') fileIn in the system workspace, select it and then do it:

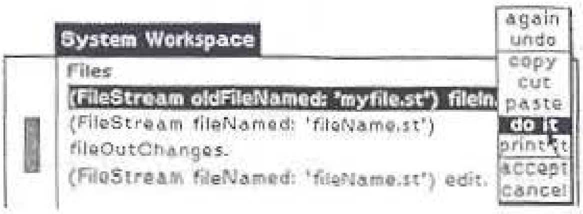

System Workspace
Files
(FileStream oldFileNamed: 'myfile.st') fileIn.
(FileStream fileNamed: 'fileName.st')
fileOutChanges.
(FileStream fileNamed: 'fileName.st') edit.
again
undo
copy
cut
paste
do it
print it
accept
cancel

Figure 2.22: Filing in from the system workspace

2.5.3 Print Out

Use print out exactly like file out, but choose the print out option from the middle button menu. A printable file will appear in your current directory. At UNC, print this file with:

```
ptroff -ms -Pprinter filename
```

2.5.4 Snapshots

One way to save the changes you have made is to select save from the main menu. You will be asked to enter a file name for the image. The next time you fire up Smalltalk, you can restore the system as you left it by entering

```
st fileName
```

This saves a snapshot of the present state of the system. Use this sparingly, since a snapshot requires 2 megabytes or more of space. Reliability problems have occurred with snapshots at UNC.

2.6 Leaving Smalltalk

2.6.1 Suspend

Suspend acts like *Control-Z* in Unix: when you return to the system by typing fg, Smalltalk will resume.

▷ Select suspend from the main menu.

▷ With the left button, select yes from the confirmer menu which appears.

You will be in UNIX.

▷ Return to Smalltalk by typing fg.

Sometimes garbage appears on the display when you return to Smalltalk. In this case, use restore display from the main menu.

You can also suspend Smalltalk at any time by pressing the L2 key. This is a short cut since no confirmer will pop up.

2.6.2 Quit

▷ To exit Smalltalk, select quit from the main main menu. Then select quit without saving from the confirmer menu which appears.

Chapter 3

Classes

3.1 Overview

This chapter presents the classes provided in the Smalltalk system. It demonstrates how you can use existing classes as tools to write efficient, effective code. The organization of these classes is described to give you some idea of how to find and use the tools you need. One important category of classes, the Collections, is discussed in detail.

3.2 Programming with Classes

The Smalltalk system includes a very large set of predefined classes. Each one has methods which specify the behavior of instances of that class. This set of classes is a taxonomy of data types which you should use as tools to help you program.

The first task of programming is to specify your application in terms of objects. Choose objects to represent important entities in your program. Think of the objects the users will have in mind when they use it; these are good candidates for Smalltalk classes. Traditional data types such as trees and graphs are also good candidates. Many objects of this kind are provided as built-in classes.

Next, specify the behavior of these objects. The behavior will be implemented as Smalltalk methods.

The next task is to choose (or build) classes to represent your objects. Every object will be an instance of some class. To do this, you need to know where to find particular types of classes and how to find out what a class does.

3.3 The Range of Built-in Classes

Smalltalk provides a wide variety of classes. Some of these are magnitude classes, like Number, Date, and Time; collections like Array, LinkedList, and Text; and graphical objects, like Rectangle, Arc, and Cursor. This information is organized in two ways:

- To aid the programmer, classes are grouped into *categories*
- Classes are organized in a strict *hierarchy*.

Categories and the hierarchy both ease the task of finding appropriate classes. The correspondence between these two is fuzzy: classes grouped in a category do not neccesarily match portions of the hierarchy.

3.4 Categories

For the convenience of the user, classes are grouped into categories. All categories are listed in the top left subwindow of the browser window. One of the best ways to locate useful classes is to choose a category which seems relevant to your needs and then browse through the classes in that category. These are most commonly used categories:

```
Numeric-Magnitudes
Numeric-Numbers
Collections-Abstract
Collections-Unordered
Collections-Sequenceable
Collections-Text
Collections-Arrayed
Collections-Streams
Graphics-Primitives
Graphics-Display
Graphics-Paths
Kernel-Objects
Files-Streams
Unix-Interface
```

Figure 3.1: Some Useful Categories of Classes

3.5 The Hierarchy

Every Smalltalk class has some position in the class hierarchy. The hierarchy is a tree structure: every class has one immediate *superclass* and any number of *subclasses.* Classes which are related to each other are often close together in the hierarchy.

The main purpose of the class hierarchy is to exploit common characteristics of objects. The hierarchy is partly determined by implementation issues, so its structure does not always make sense from a conceptual standpoint.

Classes inherit methods and variable declarations from all their superclasses. By default, all methods of a class are inherited by all its subclasses. A subclass may explicitly override a method specified in one of its superclasses or it may add new methods. Thus, you cannot infer the behavior of an object by the methods in its class alone; you have to take into account its superclasses.

The class Object is at the top of the hierarchy. All other classes are subclasses of Object. Object provides fundamental methods which are inherited by all classes. These include copy (to make a copy of an instance), and = and == (to compare instances).

Inheritance of methods is very important because if you choose classes judiciously, the amount of code you will need to write can be very small. For example, to create block diagrams of houses you could define a class House from scratch. Or you could use the existing class Rectangle. To use Rectangle, create a subclass House and add an extra point to mark the top of the roof. Then add or modify methods provided by Rectangle as needed to deal with the roof. In fact, *all* the code you write for House will specify ways in which it differs from Rectangle.

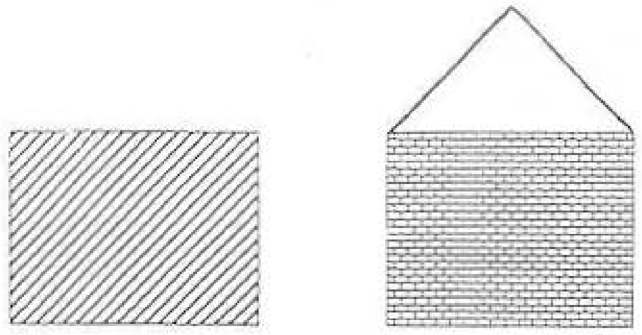


Figure 3.2: A Rectangle and a House

3.6 An Example Hierarchy

Suppose you are writing a program to draw circles and ellipses. You could define Circle and Ellipse as independent classes:

```
class: Circle
   variables: center, radius
   methods: moveTo ...
class: Ellipse
   variables: center, radius1, radius2
   methods: moveTo, ...
```

The declarations have similarities:

- the variable center is declared in both
- the method moveTo is declared in both and it can be implemented the same way

The following class hierarchy would be useful:

```
super-class: CenteredObjects
      variable: center
      methods: moveTo.
   sub-class: Circle
      variable: radius
      methods: ...
   sub-class: Ellipse
      variable: radius1, radius2
      methods: ...
```

This structure is less redundant and easier to maintain than the previous one. The code has been split into sections and duplicate parts are factored out to a higher level. Subclasses Circle and Ellipse can use the instance variables defined in any of their superclasses as well defining new ones.

To program in Smalltalk, specify the important objects in your applications and choose classes to represent these objects. Then modify, add, and remove methods until instances of the classes behave as you intend. You may have to create new classes; if so, they probably will include temporary variables which are instances of existing Smalltalk classes. Whenever possible, use the classes provided instead of reinventing them.

To select the one of several closely-related classes that is best suited to an application can be difficult. The best way to choose between classes is to examine the methods provided. What follows is a short guide to a commonly used set of categories: the Collection classes.

3.7 Collection Classes

In many languages, when you need to use a list of objects, you declare an array or program a linked list. In Smalltalk, you can choose one of the collection classes to represent your list.

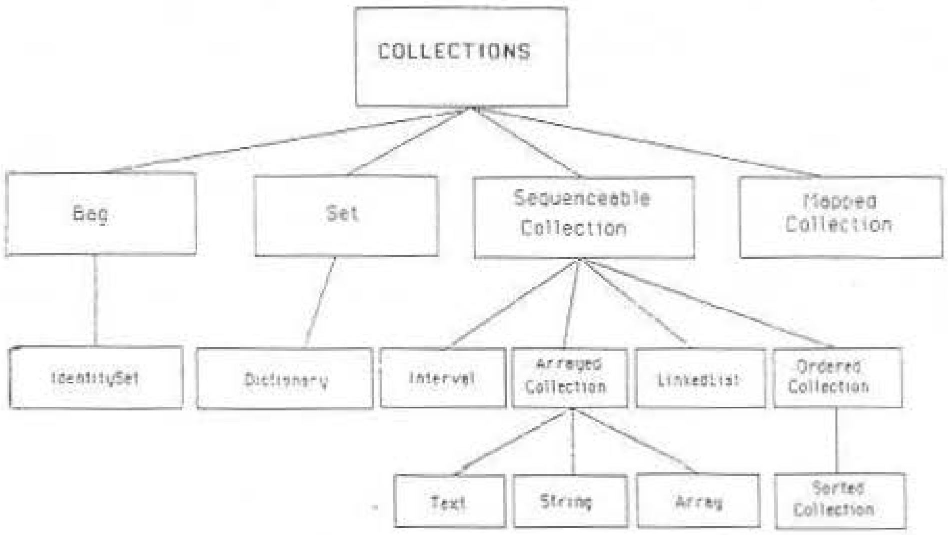


Figure 3.3: Hierarchy of the Collection Classes

If you choose a class which is far down in the hierarchy, you get the advantage of inheriting a lot of methods from its superclasses. If the class you choose inherits an inappropriate method, you have the options of not using it or of replacing it by adding a method of the same name in your class. For example, suppose you want to represent a list of students in alphabetic order. You would like to be able to insert and delete students, while maintaining alphabetic order. Look at the classes

SequenceableCollection
OrderedCollection
SortedCollection

Which class would you choose? Notice that SortedCollection and OrderedCollection are both subclasses of SequenceableCollection.

You could select SequenceableCollection itself. But would you gain more capabilities if you chose one of its subclasses?

If you use an instance of either SortedCollection or OrderedCollection to represent and manipulate your list, you will be able to use all methods defined for SequenceableCollection. Which of these two classes would be most appropriate to your goal? To determine this, look at the methods defined on OrderedCollection and SequenceableCollection and decide which one contains the most useful methods.

If you choose SortedCollection, your list will be kept sorted, since the class you have chosen provides this facility. By judicious selection of classes, you can avoid writing routines and keep your Smalltalk code very, very short.

This figure depicts decisions you will want to make to choose a class to represent your list:

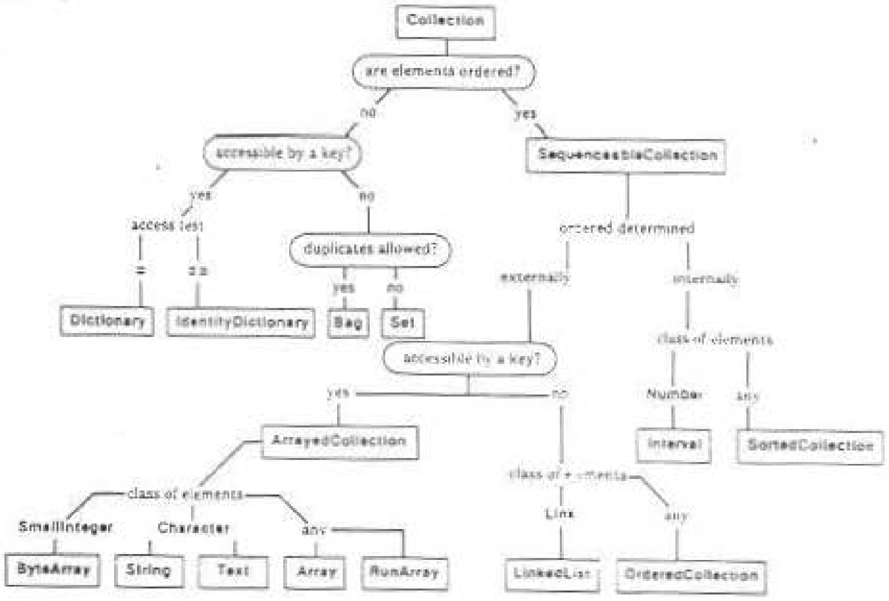


Figure 3.4: Decision Tree of the Collection Classes, reproduced from [1]

3.8 Choosing Effective Objects, Classes, and Methods

Objects are like modules in procedural languages, so they should conform to the same principles as modules:

- Hide information and implementation details.
- Encapsulate specifications that are likely to change.
- Be as general as possible and permit reuse.

For each class, determine a set of methods which intuitively apply to that object. All other methods should be considered *private*. Unfortunately, all Smalltalk methods are global so there is no way to enforce hiding. Put such methods in a protocol called private.

For example, appropriate operations for a tree are: get the root, get the value of each node, and traverse in pre-order, in-order, and post-order. Inappropriate operations for a tree are "get the value of an internal pointer" or "traverse in stored order". These depend on the implementation and have no place in the concept of a tree.

Reuse method names for similar objects. For example, Smalltalk collections define the method do: to traverse all their members. Use the do: to express the same concept in your data structure, instead of making up a new name. You will need to remember less.

Chapter 4

Programming

4.1 Overview

This chapter describes how to write classes and methods.

The main components of the programming task are:

- define classes using *class templates*
- write *instance methods* to access and manipulate instances of the classes
- write *class methods* to create these instances
- debug

The first part of the chapter describes the programming task with an example. The illustrations show how to use the Smalltalk browser for programming.

The second part of the chapter describes the programming language in detail.

The program code for the example is listed in Appendix D.

4.2 Defining a Class

4.2.1 The Example

Suppose you would like to draw many dots on the screen:

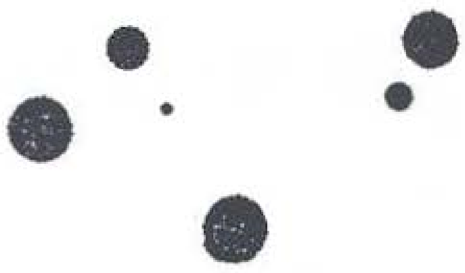


Figure 4.1: The example: dots drawn on the screen

You can define a class to do this task. Each dot will be an instance of the class. A dot can be created, assigned a size and a position, and displayed.

The name of the class will be Dot. Class names always begin with a capital letter.

4.2.2 Specification

As with any programming task, before you start programming, clarify what you intend to do. Determine the data structures you need and the operations you need to perform on the data.

Then define classes to implement the data structures and operations. This step is explained in this chapter.

Once the classes are defined, you can run your program to create new objects according to the definition you have provided. These objects will respond to the messages sent to them.

4.2.3 Defining a New Class

To define a new class with the browser, you may wish to create a new *category* to contain the class. This is done for clarity and documentation purposes only.

▷ Create a category with the browser:

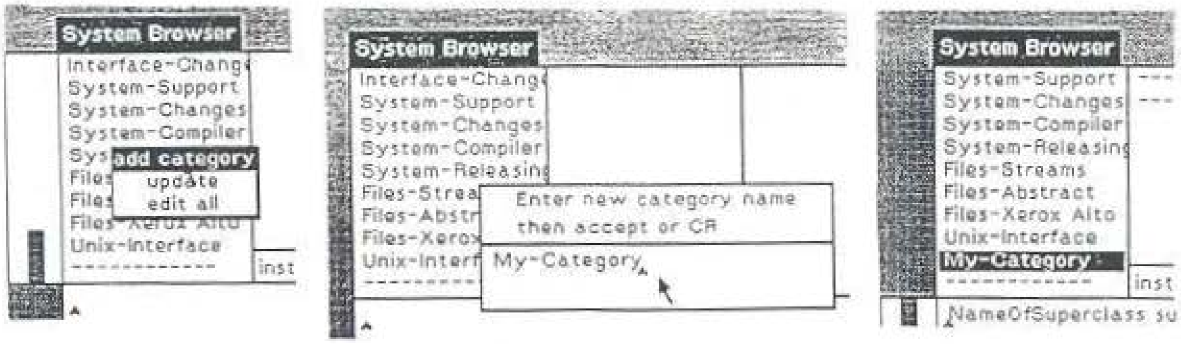


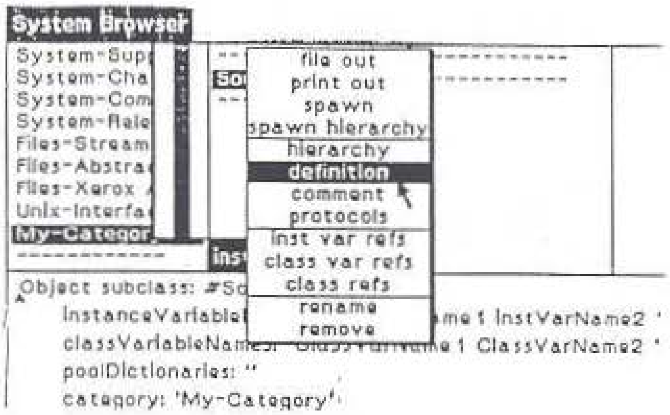
Figure 4.2: Creating a new category

You may wish to review in chapter 2 how to open a browser.

Now define the new class.

You can describe a Dot by its center and its radius. The center is a point on the screen, and the radius is a number. Only one variable is needed for the center (not two for the x and y coordinates), since Smalltalk provides Points. Thus a Dot will have two instance variables: center and radius.

▷ Get the *class definition template* with the middle mouse button in the class pane of the browser:



```
Object subclass: #Sc
    instanceVariable                    me1 InstVarName2 '
    classVariableNam              e1 ClassVarName2 '
    poolDictionaries: ''
    category: 'My-Category'
```

Figure 4.3: Getting the class definition template

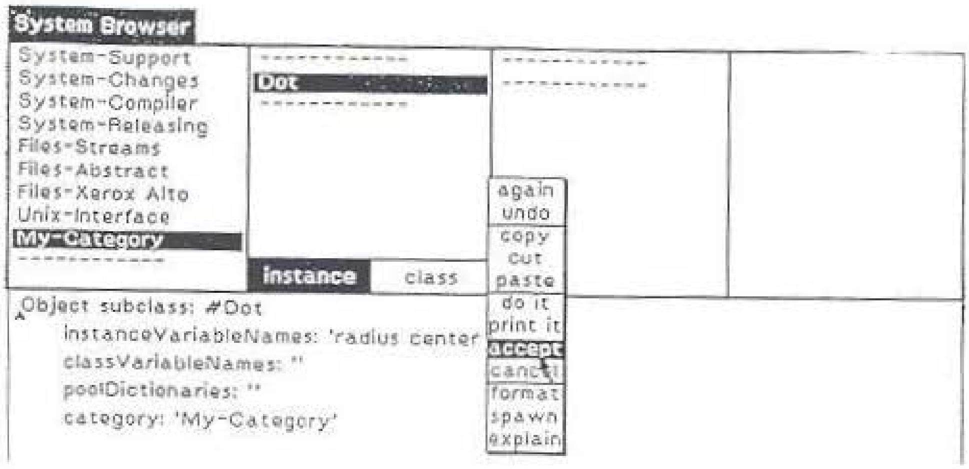
This is the class definition template:

```
NameOfSuperclass subclass: #NameOfSubclass
    instanceVariableNames: 'instVarName1 instVarName2'
    classVariableNames: 'ClassVarName1 ClassVarName2'
    poolDictionaries: ''
    category: 'My-Category'
```

▷ Modify it as follows:

```
Object subclass: #Dot
    instanceVariableNames: 'radius center'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'My-Category'
```

and accept it with the middle mouse button:



```
Object subclass: #Dot
        instanceVariableNames: 'radius center
        classVariableNames: ''
        poolDictionaries: ''
        category: 'My-Category'
```

Figure 4.4: Accepting the class definition

Now the Dot class is defined. It is a subclass of Object, so any instance of a Dot will also respond to the methods defined for Object.

Please note that this is an example only. Should you ever want to program a dot, define it as a subclass of Circle to get most of this behavior with only a little bit of effort.

4.2.4 Testing the Class Definition

Now write a short test program that uses this new class. Such programs can be written in a workspace.

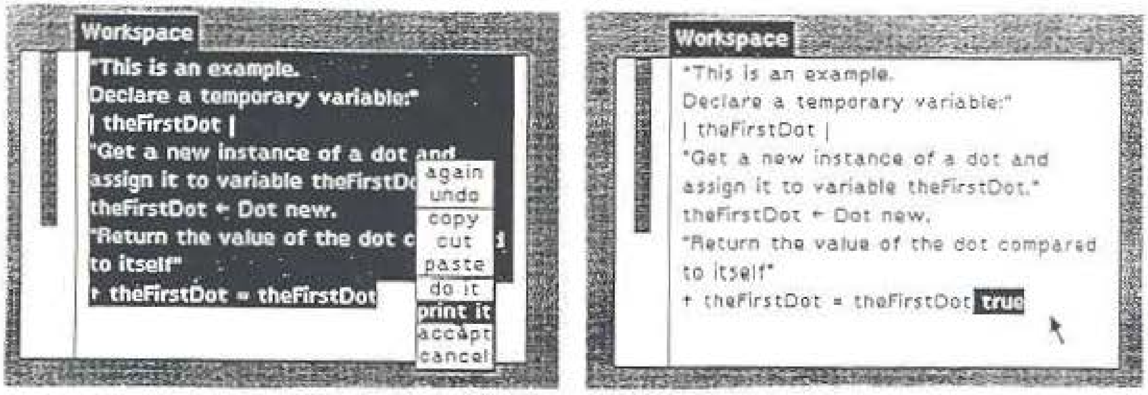
▷ Open a workspace. (See chapter 2 for a description of how to open a workspace.)

▷ Type in this program. (You may leave out the comments shown in quotes.)

```
"This is an example.
Declare a temporary variable:"
| theFirstDot |
"Get a new instance of a dot and
assign it to variable theFirstDot."
theFirstDot ← Dot new.
"Return the value of the dot compared to itself"
↑ (theFirstDot = theFirstDot)
```

Type in the *assignment* operator ← as an underscore and the *return* operator ↑ as the up-arrow or hat symbol. The period separates statements (much like the semicolon in Pascal).

▹ Select the text with the left mouse button and print it. Smalltalk will execute the program and display the return value:



```
"This is an example.
Declare a temporary variable:"
| theFirstDot |
"Get a new instance of a dot and
assign it to variable theFirstDot."
theFirstDot ← Dot new.
"Return the value of the dot compared
to itself"
↑ theFirstDot = theFirstDot true
```

Figure 4.5: Executing the program and printing the result

As expected, the return value printed by Smalltalk is true, because the variable theFirstDot was compared with itself. Press the delete key to erase the result.

Note that the new function and the comparison (=) are already available for Dot because both are defined in the superclass of Dot, Object, and a class inherits all methods of its superclasses.

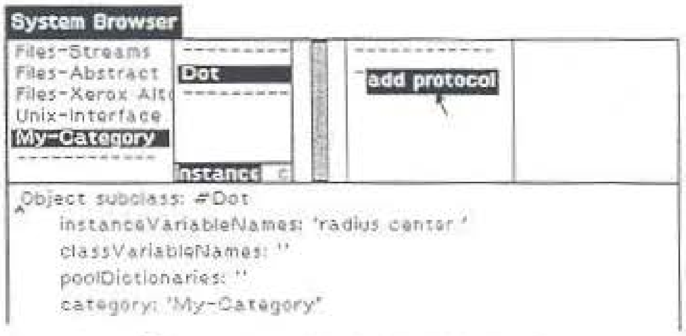
4.3 Defining Operations

4.3.1 Protocols

We will specify these operations for an instance of a Dot:

- accessing
  - change the radius
  - change the position of the center
  - ask the dot for its radius
  - ask the dot for position of its center
- calculating
  - calculate the diameter
  - calculate the area
- displaying
  - display it

Remember that groups of operations (like accessing, calculating and displaying) are called protocols. Here is how you define protocols. Make sure the instance box in the class pane is highlighted.



```
Object subclass: #Dot
    instanceVariableNames: 'radius center '
    classVariableNames: ''
    poolDictionaries: ''
    category: 'My-Category'
```

Figure 4.6: Defining protocols

Protocols are for documentation and convenience. If you have many methods, you can find methods more easily if you group them and give them meaningful names.

4.3.2 Writing an Instance Method

▹ To write an instance method, replace the *method template* in the browser. The template is:

```
message selector and argument names
    | temporary variable names |

    statements
```

▹ This short method replaces the radius of an instance with a new value:

```
radius: aNumber
    "Replace my radius with aNumber"
    radius ← aNumber
```

▹ Use the accept option of the middle button menu to compile and store this method (this will take only a few seconds).

A method behaves much like a procedure:

- It has a name, **radius:**. (The colon is part of the name, and indicates that an argument follows.)
- It has arguments. This example has the single argument **aNumber**.
- It can have temporary (local) variables. They are named between vertical bars. Temporary variable names start with a lowercase letter. This example has no temporary variables.
- It has a body. The body of this method has only one assignment statement.

- It may return information to the caller.

All instance variables, like radius, are *visible* from all instance methods of the class, much like global variables are visible in procedures in conventional programming languages. Instance variables in one class are not visible from other classes, *except* for subclasses of that class.

There is one important difference between methods and subroutines: methods belonging to different classes can have the *same name*, For example, the class Circle also has a method called **radius:**, which is different from this one.

▷ As an exercise, write the method center: to move the center of a dot.

4.3.3 Testing an Instance Method

Here is an easy way to test this instance method:

▷ Write the following program in the workspace:

```
| x y |
"Assign a new instance of dot to both x and y."
x ← Dot new.
y ← Dot new.
"Invoke the method radius: with argument 10 upon x."
x radius: 10.
"Change radius of y to 20."
y radius: 20.
"Return x compared to y."
↑ ( x = y )
```

▷ Execute the code. The result should be false.

4.3.4 Writing a Method with Several Parameters

▷ Here is a method to replace both the radius and the center of a dot in one call:

```
center: newCenter radius: newRadius
    "Replace my radius by newRadius and my center by new-
    Center"
    center ← newCenter.
    radius ← newRadius
```

4.3.5 Self

The previous example contains two assignments. But two methods (the **radius:** and the **center:** method) already perform this assignment. Why not call them instead?

The variable self refers to the object to which the current message was sent. For example:

```
"Suppose aDot and anotherSpot are instances of the class Dot."
aDot center: a radius: b
anotherSpot radius: b
```

Inside the method **center:radius:**, self refers to the object aDot, and within the method **radius:**, self refers to the object anotherSpot.

This means that the method **center:radius:** can be rewritten as follows:

```
center: newCenter radius: newRadius
    "Replace my radius by newRadius and my center by newCenter.
    Use methods radius: and center: to perform the change.
    Call these methods on myself.
    This method is equivalent to that of the previous section."
    self center: newCenter.
    self radius: newRadius.
```

Note that self is an *implicit parameter* of any method call.

4.3.6 Inspecting an Instance

Here is another way to test the effect of a method, which is better than printing its result.

▷ Write the following code in the workspace, and do it.

```
| aDot |
aDot ← Dot new.
"Set radius to 10 and center to the point 100@200.
(The @ sign is the Smalltalk operator to form points.)"
aDot center: 100@200 radius: 10.
aDot inspect
```

The aDot inspect instruction opens a window to access the instance variables of the dot:

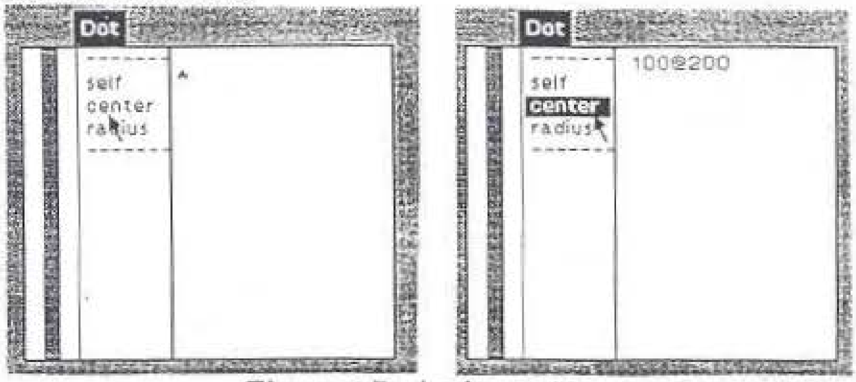


Figure 4.7: An inspector

▷ You can select each of the instance variables with the left mouse button and inspect them too. You even can inspect self, but this gets you another copy of the same inspector.

▷ To get rid of an inspector, use the right mouse button close function.

4.3.7 Returning a Value

Here is a method with no parameters that returns a value:

```
center
    "Return my center"
    ↑ center
```

▷ Write a short program to test this method in conjunction with center:.

4.3.8 A Method to Display a Dot

▷ This method displays a dot. Put it in the displaying protocol. It is based on existing Smalltalk software.

```
display
    "Display myself on the screen
    The variable dotBitmap holds a bitmap for the dot,
    and is copied to the bitmap display of the workstation."
    | dotBitmap |
    "Create the bitmap of size radius*2"
    dotBitmap ← Form dotOfSize: ( radius * 2 ).
    dotBitmap
        displayOn: Display
        at: center
        rule: Form paint
```

▷ Now try it with the following program in the workspace:

```
| theDot |
theDot ← Dot new.
theDot center: 100@100 radius: 20.
theDot display.
theDot center: 200@200 radius: 50; display.
```

The last statement is shorthand; this is called *cascading* a message, and it applies the display call to the same object as the previous call.

4.3.9 Creating New Instances

The standard way to obtain a new, uninitialized object is to send the new message to the class, as in Dot new. Normally, the first operation you perform on any new instance is to initialize some or all of its instance variables.

The *class methods* can be used to create and initialize objects in one step.

Class methods:

- initialize new instances
- cannot reference instance variables directly, as instance methods can
- are sent to the name of the class, as in Dot center: 20@20 radius: 10.

▷ Select the class box in the class pane. Then define protocols and methods just as for instance methods.

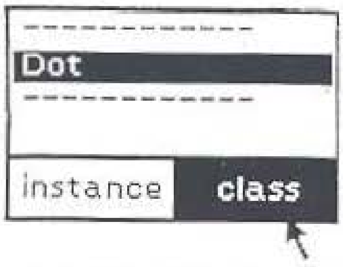


Figure 4.8: Selecting class methods

▷ Enter this class method:

```
defaultShape
    "Return a dot with default size and position"
    | x |
    "Get a new dot."
    x ← Dot new.
    "Initialize it with some default values."
    x center: 100@200 radius: 10.
    ↑ x
```

This method is used as follows:

```
| blob |
blob ← Dot defaultShape
```

4.4 Subclasses

Suppose you want to draw flowers:

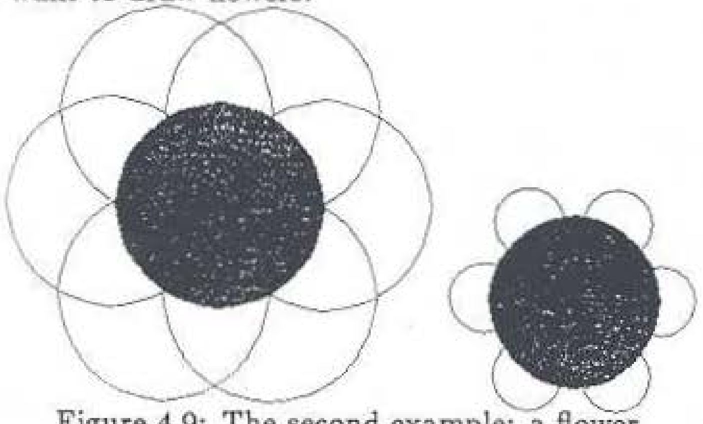


Figure 4.9: The second example: a flower

This example has many features in common with the dot example:

- A flower has a center and the radius of the central dot
- It needs similar methods to set and request the value of the instance variables center and radius

But some additional features are needed:

- The length of the petals must be specified
- The display of the flower is different, but similar to the display of the dot.

4.4.1 Defining a Subclass

▷ Define Flower as a subclass of Dot:

```
Dot subclass: #Flower
   instanceVariableNames: 'petalLength'
   classVariableNames: ''
   poolDictionaries: ''
   category: 'My-Category'
```

This will *add* the instance variable petalLength to the instance variables center and radius of Dot. In this respect, it is the same as declaring:

```
Object subclass: #Flower
   instanceVariableNames: 'radius center petalLength'
   classVariableNames: ''
   poolDictionaries: ''
   category: 'My-Category'
```

But if Flower is defined as a subclass of Dot, then also *all methods* of Dot are available for Flower, such as **center, radius, center:, radius:, center:radius:, display** and **defaultShape**. Thus, you can *build* on existing software, by adding and replacing features.

4.4.2 Defining New Methods in a Subclass

▷ The following methods are also needed for a Flower:

```
petalLength: newLength
   petalLenth ← newLength

petalLength
   ↑ petalLength
```

New methods, with different names from the superclass methods, can be added as desired.

4.4.3 Overriding a Method

You can also write a method in a subclass to *override* a method in the superclass. For example, the **display** method of Flower is different: it should also display the petals. Here is the new **display** method. It uses the standard Smalltalk object Circle to draw the petals.

▷ Type this method:

```
display
    "Display myself on the screen.
    Temporary variables: petal points to a circle."
        | petal c |
    "Display the petals. Use the Circle object to draw little
    circles. Get a (new) circle, and set the thickness of the
    line to 1. The radius of the circle is the petal length."
    petal ← Circle new.
    petal form: (Form dotOfSize: 1).
    petal radius: petalLength.
    "Draw six petals with this loop."
    0 to: 300 by: 60 do: [ : angle |
        "Calculate the center of the petal"
        c ← (angle degreesToRadians cos) @
                (angle degreesToRadians sin).
        c ← c * radius + center.
        "Set the center of the petal"
        petal center: c.
        "and display it."
        petal display ].
    "Now draw the dot inside the flower."
    aDot ← Dot new.
    aDot center: center radius: radius.
    aDot display
```

This method replaces the method of the superclass for all Flower objects. The display method defined on Dot is still available for instances of the Dot class, but it is hidden for instances of the Flower class.

4.4.4 Use of super

super permits calling methods of a superclass, which would be otherwise redefined by the class.

It is a variant of self, but the method is looked up in the superclasses only, instead of this class and the superclasses.

▷ Here is an example of the use of super. The method in the last example, which displayed a Flower, can be rewritten more efficiently as:

```
display
    "Display myself on the screen.
    Temporary variables: petal points to a circle"
        | petal c |
    ...same as before ...
    "Now draw the dot inside the flower."
    super display
```

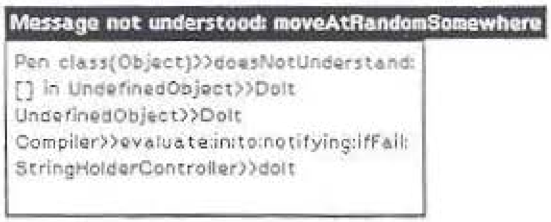
The super display statement calls the display method of Dot. Thus, the new Flower display method is really a *refinement* of the Dot method: it *adds* some statements to the original method (it adds the statements that draw the petals).

The use of super is appropriate when a method in a subclass should perform functions similar to those are already programmed in a superclass. You only need to write code to do the additional functions.

4.5 Debugging

4.5.1 Run time errors

Whenever Smalltalk detects a runtime error, a notifier appears:

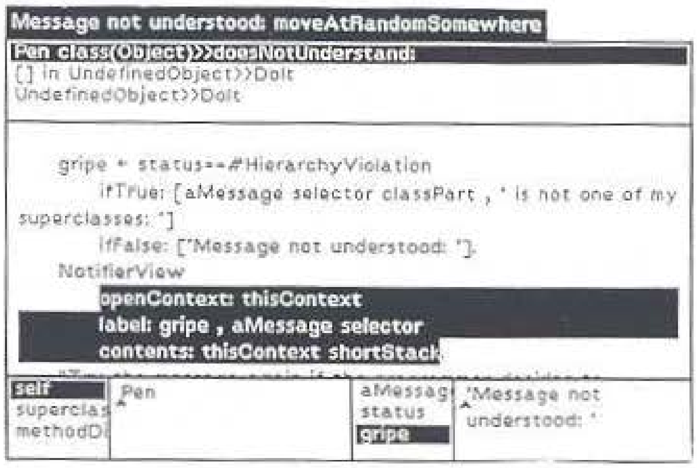


```
Message not understood: moveAtRandomSomewhere
Pen class(Object)>>doesNotUnderstand:
[] in UndefinedObject>>DoIt
UndefinedObject>>DoIt
Compiler>>evaluate:in:to:notifying:ifFail:
StringHolderController>>doIt
```

Figure 4.10: A notifier for a runtime error

Using the middle button menu, you can either proceed (which tries to ignore the error) or debug. You also can use the close option of the right button menu to abandon the task.

If you select debug, a debugger window appears:



```
Message not understood: moveAtRandomSomewhere
Pen class(Object)>>doesNotUnderstand:
[] in UndefinedObject>>DoIt
UndefinedObject>>DoIt

    gripe ← status==#HierarchyViolation
        ifTrue: [aMessage selector classPart , ' is not one of my
superclasses: ']
        ifFalse: ['Message not understood: '].
    NotifierView
        openContext: thisContext
        label: gripe , aMessage selector
        contents: thisContext shortStack

self        Pen                 aMessag   'Message not
supercla                        status     understood: '
methodD                         gripe
```

Figure 4.11: A debugger window

The first subwindow shows the stack of calls. There is one line for each call and the most recent call is first. You can move up and down the stack to select the method that interests you.

The middle button menu in this subwindow is:

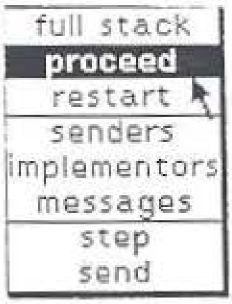


Figure 4.12: Middle button menu in the top subwindow

Some of the options are:

proceed

Continues the execution.

single step

Executes instruction by instruction.

full stack

Shows the *full* stack of calls, not only part of it.

The second subwindow shows the source code of the selected method. The current instruction is highlighted.

If you discover an error you can change the source code, accept it, and then select proceed on the middle button menu of the top subwindow.

The third subwindow consists of an inspector of the instance variables (left side) and of the temporary variables (right side).

4.5.2 Inserting breakpoints

A breakpoint is the instruction:

```
nil halt
```

Whenever it is executed, the debugger appears.

4.5.3 Loops

If your program is in a loop, hit control-C. A notifier will appear, and you can enter the debugger.

You can use the L2 key to suspend Smalltalk and enter Unix (see section 2.6.1) if control-C doesn't help.

4.6 Method syntax

4.6.1 Overview

This section describes method syntax in detail. The definitions are practical rather than rigorous. Use this section as a reference. For more information, see [1].

Four topics are covered:

- constants and variables
- types and precedence of message expressions
- arithmetic and boolean operations
- arrays
- common control structures, including if-then-else and while

4.6.2 Constants

These are some of the *constants* that can appear in Smalltalk programs:

example	type
10	decimal
10.3	decimal
10.3e-4	decimal with exponent
-6	decimal, negative
'hello'	string
'don''t'	string with ' in it
$x	one character (the x)
true, false	the boolean true and false value
nil	the value of an uninitialized variable
#(10 20 30)	an array constant with 3 elements
#center:	a symbol (for example a method name)

Table 4.1: Smalltalk constants

4.6.3 Variables

Smalltalk variable names are alphanumeric. The first letter of a variable indicates its scope:

- First letter lowercase: *local variables*, like instance variables or temporary variables.
- First letter uppercase: *global variables*, like class variables or class names.

4.6.4 Types and Precedence of Message Expressions

Smalltalk statements are message expressions. Here are the three types of simple message expressions (the keyword message is illustrated twice):

sample expression	type	arguments
aDot display	unary	none
radius * 3.14159	binary	3.14159
aDot radius: 20	keyword	20
aDot center: aPoint radius: aNumber	keyword	aPoint and aNumber

Table 4.2: Message types

All expressions are a combination of these types of messages: unary, keyword and binary. Their characteristics are:

Unary messages

Unary messages have no arguments.

The name of a unary message is alphanumeric, starting with a lowercase letter.

Unary messages have the highest precedence.

Binary messages

Binary messages have exactly one argument.

The name of a binary message consists of one or two special characters, like + / * ~ > < @ % | & ? and -.

Binary messages have intermediate precedence.

Keyword messages

Keyword messages can have any number of arguments.

Each argument is indicated by a keyword and a colon (:). Keywords are alphanumeric, starting with a lowercase letters.

Keyword messages have the lowest precedence.

Use *parenthesis* to alter the precedence or to clarify your intentions. Indenting an expression may also add to the clarity. For example:

```
BitEditor
    openScreenViewOnForm: (Form fromDisplay)
    at: 0@0
    magnifiedAt: 100@100
    scale: 8@8
```

Within the same precedence, expressions are interpreted *strictly from left to right*. For example index + offset * 2 is the same as (index + offset) * 2 .

All messages can be *cascaded* with a semicolon. That is, several messages can be sent to the same object in just one statement:

```
aDot radius: 10; center: 20@20; display.
```

is the same as

```
aDot radius: 10.
aDot center: 20@20. aDot display.
```

Variables *point* to objects, they do not *contain* objects. For example:

```
"Create a new rectangle, and make 'a' point to it:"
a ← Rectangle origin: 0@0 corner: 100@120.
"Make 'b' reference the same rectangle:"
b ← a.
"Modify the rectangle, moving its origin:"
a moveTo: 10@20.
```

In this example, both b and a point to the same object. The last line modifies the object, so after this operation, b will have moved too!

If you want to assign a copy of an object to a variable, use:

```
b ← a deepCopy.
```

Now a and b point to different objects, and whatever you do with a will not affect b.

4.6.5 Common Operations

All numeric objects accept the four *arithmetic* operations (messages) +, −, * and /, and the *comparison* operators <, >, >=, <= and =.

All objects can be compared using =, to see if they have *identical values*.

Two objects can also be compared using ==: this test will succeed only if the variables which name these objects point to exactly the same memory location.

To see whether a variable x has been initialized, that is, has value nil, use x notNil.

The *boolean* operators are:

operator	meaning
&	and
|	or
not	negation

Table 4.3: Boolean operators

Be careful: since not is a unary operator, it has to go *after* the expression, so that

```
( a | b ) not
```

really means

not(a or b).

All parts of boolean expressions are evaluated. See the control structures and: and or: for variants that evaluate only the necessary parts of the expression.

Arrays are created with the Array new: message:

```
"Assign an array of 20 empty elements to x"
x ← Array new: 20.
```

Use at: to refer to an element and at:put: to replace the value at an element:

```
"Assign the 5th element of x to y"
y ← x at: 5.
"Replace the 7th element of x by z"
x at: 7 put: z
```

The message size returns the size of an array. For example, return true if the i-th element of x is positive, false if it is negative or the index i is out of bounds:

```
↑ ( i > 0 and: [ i < x size and: [ ( x at: i ) > 0 ]] )
```

The elements of an array can contain any object: numbers, dots, strings, characters, arrays, etc. Different elements of a single array can contain different object types.

Many other types of *collections* are available in Smalltalk: sets, ordered collections, dictionaries, bags, etc. See [1] or the Smalltalk code for more details.

Strings are a particular form of array. You can use at: to access a particular character in a string. Concatenate strings with

```
bothStrings ← string1 , string2.
```

4.6.6 Control Structures

Smalltalk has many *control structures*. Creating new ones is easy.

Here are some control structures similar to those found in other languages. A *block* is a sequence of statements surrounded by brackets [and].

```
condition ifTrue: [ statements ].
condition ifFalse: [ statements ].
condition ifTrue: [ statements ] ifFalse: [ statements ].
[ condition ] whileTrue: [ statements ].
[ condition ] whileFalse: [ statements ].
1 to: 10 do: [ : index | statements ].
```

Iteration with index varying from 1 to 10

```
1 to: 100 by: 5 do: [ : index | statements ].
```

Iteration with index set to 1, 6, 11 ...96

```
n timesRepeat: [ statements ].
```

Repeat statements n times

```
condition1 and:[ condition2 ]
```

and operation; evaluates condition2 only if condition1 is true

```
condition1 or:[ condition2 ]
```

or operation; evaluates condition2 only if condition1 is false

Example: Return the maximum of a and b.

```
a > b ifTrue: [ ↑ a ]
      ifFalse: [ ↑ b ].
```

The do: loop is useful for processing all members of a collection. It works with arrays, sets, bags, lists, and other collections:

```
aCollection do: [ : element |
   ...process element ]
```

The variable element points to each element of the collection as the collection is scanned.

Example: find the maximum in the array x:

```
max ← x at: 1.
x do: [ : element |
      element > max ifTrue: [ max ← element ] ]
```

Appendix A. Reference

Appendix A

References

A.1 Overview

This section describes books and articles you may wish to consult.

A.2 Books

The Smalltalk language is described in the "Blue Book":

1. Adele Goldberg, David Robson, *Smalltalk-80, The Language and its Implementation*. Addison-Wesley, 1983.

This book starts from ground up. The first chapters define concepts like objects, methods, messages and classes.

It contains descriptions of most Smalltalk classes:

- Numbers: integer, floating point, fraction, random numbers
- Character and string

- Collections: array, set, interval, bag, dictionary, ordered collection, etc.
- Stream and file stream
- Files
- Process management: process, semaphore, shared queue
- The classes Object and UndefinedObject
- Class objects: metaclass and class
- Graphical objects: point, rectangle, arc, circle, curve, line, linear fit, spline
- Display objects: forms, display screen, cursor

Topics not covered include mouse, menus, fill in the blanks and Smalltalk windows.

The "Orange Book" describes the programming tools:

> **2.** Adele Goldberg, *Smalltalk-80: The Interactive Programming Environment.* Addison-Wesley, 1984.

If you have used Smalltalk you will be familiar with most of the contents of this book, but it contains many interesting hints and details. Many facilities, such as the text editor, are covered in depth.

Another introduction to Smalltalk is:

> **3.** Ted Kaehler, Dave Patterson, *A Taste of Smalltalk.* W. W. Norton & Co., 1986.

It uses examples, contains formal definitions, and explains the use of many classes. It also has special sections on the text editor and how to read source code.

A.3 Journal Articles

A source of good articles on Smalltalk is

4. *BYTE 6, 8.* (August 1981).

It contains introductory articles and also some more advanced ones, like *Design Principles behind Smalltalk.* Several articles discuss implementation issues.

Another worthwhile article in the same issue discusses control structures. It demonstrates how to pass blocks as parameters and how to write your own control structures.

A collection of papers on Smalltalk-80 is:

5. Glenn Krasner, ed., *Smalltalk-80: Bits of History, Words of Advice.* Addison-Wesley, 1983.

Appendix B. Reading Smalltalk Source Code

Appendix B

Reading Smalltalk Source Code

Source code for all Smalltalk objects is (usually) visible in the Smalltalk environment. It is a good example of

- object oriented design
- programming methodology
- how to use classes and methods effectively

Unfortunately, it is often poorly documented.

Before you begin to program, search through this code to see if it includes an object you can use. Smalltalk has commands and features to help you find objects. Most searching is done with the browser.

The **senders** menu option in the methods pane permits access to all methods that call the selected method.

The **implementors** menu option in the methods pane gives access to all methods with the same name as the selected method.

The explain menu option in the editor points to classes, categories and methods.

Here is a strategy for reading Smalltalk source code with the browser:

- Take a look at the class hierarchy.
- Dig into a category. Select some class that seems interesting.
- Read the class comment.
- Read the class definition. Compare.
- Glance at the class methods. There may be examples. If so, execute them.
- Now go to the instance methods. A good starting point is the "protocols for accessing" section.
- If you don't find an instance method that you expect, look in the superclasses.

Appendix C

How to Program a User Interface

C.1 Overview

Smalltalk includes aids for programming a user interface. There are Smalltalk classes which give you access to the following entitites. At a low level:

- mouse
- cursor
- keyboard
- display
- the system transcript

and at a higher level:

- prompts

- menus
- choice

Many of these classes have been presented elsewhere in this manual. This appendix explains how to use these classes. The description is not detailed, but the information is sufficient to enable you to use the most important features.

The display screen is not discussed here. [1] contains a detailed explanation of the use of the display. Information can also be found with some effort in the Form class and its superclasses, and in the Graphics categories.

C.2 Mouse

The mouse has three buttons: red button (left), yellow button (middle) and blue button (right). The mouse buttons can be tested by

```
Sensor redButtonPressed
Sensor yellowButtonPressed
Sensor blueButtonPressed
Sensor anyButtonPressed
Sensor noButtonPressed
```

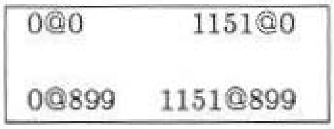
which return either true or false. You can use Sensor waitButton to wait for any of the three buttons to be pressed.

The position of the mouse is returned by the command:

```
mouseLocation ← Sensor mousePoint
```

This message returns a Point on the screen.

The coordinates of the borders of a SUN screen are depicted below:



Sensor is a global variable. It contains an instance of the class InputSensor, and lets you communicate with the keyboard and mouse of your workstation.

C.3 Cursor

The cursor is a moving image of 16 by 16 pixels on the screen. Normally, the movement of the mouse is coupled to the cursor. Whenever the mouse moves, the cursor follows it.

Management of the cursor is done through the Cursor and the InputSensor class (Sensor global variable).

The usual shape of the cursor is a leftward-pointing arrow. The class definition of Cursor also contains other shapes (arrow, arrow with star, scribbling pen, cross hairs, blank, etc).

This example shows how to read the form of the current cursor, then change it to the "write" cursor (a little pen, which appears, for example, when you do a file out), and then restore the original cursor:

```
"Get the current cursor from the Sensor."
tempCursor ← Sensor currentCursor.
"Get the write cursor and show it."
Cursor write show.
    ...
"Return to previous cursor shape."
tempCursor show.
```

The cursor can be moved to another point with the instruction:

```
Sensor cursorPoint: newPosition
```

where newPosition is a point on the screen.

You can assign any 16 by 16 Form to the cursor using the methods in the Cursor class.

C.4 Keyboard

This section explains the use of the keyboard. If you want to read strings input by the user, do not use these methods, but use the prompts instead (see section C.6).

The keyboard is accessible through the Sensor object (InputSensor class). Here are the most commonly used methods:

Sensor keyboard

> Returns one ASCII character from the input buffer. Returns false if there is no character available. (This is a non-blocking read).

Sensor flushKeyboard

> Flushes all type-ahead.

C.5 System Transcript

The system transcript is useful for displaying small amounts of information, for showing progress, and for debugging.

The system transcript is available through the global variable Transcript (class TextCollector).

Example:

displays

```
Start
List of actions
```

on the system transcript.

You can also print many other objects (like numbers, arrays, points) on the system transcript using the printString function:

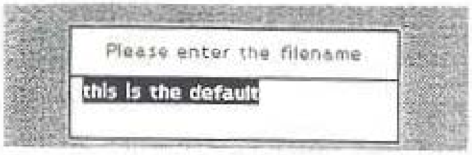
```
x ← Sensor mousePoint.
Transcript show: 'The mouse points to '; show: x printString ;
    cr.
```

Prior to using the system transcript window, it is convenient to put it the foreground:

```
Transcript refresh.
```

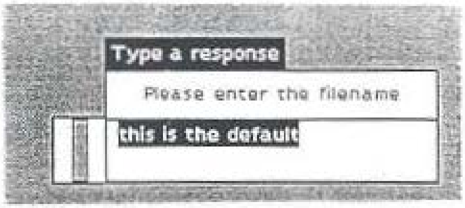
C.6 Prompts

A prompt is used to ask a question and get an answer from the user. Example:



Please enter the filename

this is the default



Type a response

Please enter the filename

this is the default

Figure C.1: A 'request' and a 'message' prompt

This is the code to produce the 'request' box:

```
| answer |
FillInTheBlank
    request: 'Please enter the filename'
    displayAt: Sensor mousePoint
    centered: true
    action: [ :answer | answer ]
    initialAnswer: 'this is the default'
```

request: gives the question to be asked. The user accepts (with the middle button menu) or presses the return key to enter his/her answer.

If you use message: instead of request: then the answer can be several lines long. In this case, the FillInTheBlank window will also have a scroll bar (see Figure C.1).

displayAt: indicates the position of the box; Sensor mousePoint is convenient, another good possibility is Display boundingBox center.

The action: block is executed when the answer has been entered. The parameter of the block receives the answer string – in this example the action leaves the answer in the temporary variable answer.

The initialAnswer: may be a null string ('').

C.7 Menus

Pop-up menus are a convenient command language. A pop-up menu looks like this:

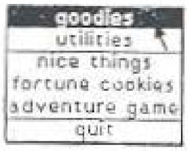


Figure C.2: A pop-up menu

The use of a pop-up menu has two parts:

- Activate the menu. This displays the menu, allows selection, and returns the index of the selected item.

Reuse menu objects; create them once at the beginning rather than each time you use one. You can also share a menu object among several places where it is used, putting it in a class variable.

Here is the code to initialize the pop-up menu of Figure C.2:

```
"Create a menu with 6 items and lines after items 2 and 5."
menu ← PopUpMenu
   labels:
'goodies
utilities
nice things
fortune cookies
adventure game
quit'
lines: #( 2 5 ).
```

There are many ways to activate a menu (see also PopUpMenu class, protocols for controlling). Here are the most common forms.

selectedItem ← menu startUp

Show the menu and wait for any button to do a selection.

selectedItem ← menu startUpRedButton

Show the menu and wait for the left button to do a selection. Other buttons will be ignored.

selectedItem ← menu startUpYellowButton

selectedItem ← menu startUpBlueButton

Similar to selectedItem ← startUpRedButton.

```
selectedItem ← menu startUp: #redButton withHeading: Title of the Menu
```

Shows a menu with a title. You also can use #yellowButton and #blueButton. A selection must be made (menu flashes if you leave it).

In all cases, the number of the menu item is returned (assigned to the variable selectedItem in the examples). 0 is returned if no selection was made, else an integer from 1 to the number of items in the menu.

You can also use ActionMenus. Instead of returning a selection, they call a routine depending on the selection. See the ActionMenu class in the Smalltalk code for an example.

C.8 Choice

A choice is a question with yes-no answer. This is how a choice looks on the Smalltalk screen:

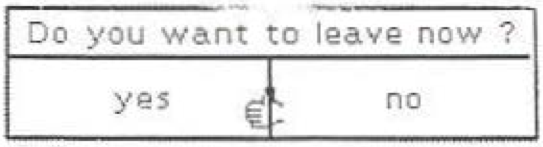


Figure C.3: A Binary Choice

The program code for this example is:

```
BinaryChoice
    message: 'Do you want to leave now ?'
    displayAt: Sensor mousePoint
    centered: true
    ifTrue: [ ...action if true ...]
    ifFalse: [ ...action if false ...]
```

Do not put a Smalltalk return command (↑) inside the ifTrue or ifFalse blocks, because the BinaryChoice will not be erased properly.

Appendix D

Sample Program

This chapter contains the sample program used in Chapter 4. The code in this appendix differs slightly from Chapter 4, and several methods have been added.

Dot

class name	Dot
superclass	Object
instance variable names	center radius
class variable names	*none*
pool dictionaries	*none*
category	Graphics-Bubbles

comment

I am a circle with a center and a radius. I can be displayed on the screen, and I appear as a black dot.

Instance Protocols For: accessing

area

```
"Return my area"
↑ 3.14159 * radius * radius
```

center

```
"Return my center"
↑ center
```

center: newCenter

```
"Replace my center by a new center"
center ← newCenter
```

center: newCenter radius: newRadius

```
"Replace both my radius and my center"
self radius: newRadius.
self center: newCenter
```

diameter

```
"Return my diameter"
↑ 2 * radius
```

diameter: newDiameter

```
"Alternative form of changing my size"
radius ← newDiameter / 2
```

```
    "Return my perimeter"
    ↑ self diameter * 3.14159
```

radius

```
    "Return my radius"
    ↑ radius
```

radius: newRadius

```
    "Replace my radius by a new radius"
    radius ← newRadius
```

Instance Protocols For: testing

containsPoint: aPoint

```
    "Return true if the dot contains aPoint.
    A circle contains a point if the distance to the center
    is less than the radius. The distance to the center is
    calculated as the length of the vector 'center - aPoint'. This
    makes use of Point arithmetic, see Point class, 'arithmetic' and
    'polar coordinates' protocols."
    ( center - aPoint ) r < radius
        ifTrue: [ ↑ true ].
    ↑ false
```

Instance Protocols For: comparing

= otherDot

```
    "NOTE. This method overrides the standard comparison method.
    In this case, dots will be equal whenever the radius is the same.
    The center may be different. This differs from the normal
    comparison (provided by the Object object) which requires
    all instance variables have the same value."
    ( radius = otherDot radius )
        ifTrue: [ ↑ true ].
    ↑ false
```

Instance Protocols For: displaying

display

```
    "Display myself on the screen"

    "Declare a temporary variable, to contain the bitmap of a dot."
    | bitmapDot |
    "Create the bitmap graphic form of a circular dot of my diameter"
    bitmapDot ← Form dotOfSize: (radius * 2).
    "Display this dot at the center. Since all 'Form' bitmaps are rectangles,
    'Form paint' indicates that the enclosing rectangle shall not be shown."
    bitmapDot
        displayOn: Display
        at: center
        rule: Form paint
"- - - - - - - - - - - - - - - - - - - -"

Dot class
instanceVariableNames: ''
```

Class Protocols For: instance creation

center: aCenter radius: aRadius

```
    "Returns a new instance of a dot, initialized with these parameters"
    "This is a short form of:
        | x |
        x ← self new.
        x center: aCenter radius: aRadius"
    ↑ ( self new ) center: aCenter radius: aRadius
```

defaultShape

```
    "Returns a new instance of Dot with default size"
    | x |
    x ← Dot new.
    x center: 100@100 radius: 10.
    ↑ x
```

Class Protocols For: examples

exampleOne

```
    "Display some dots.
    To execute, select next line and do it:
        Dot exampleOne
    "
    "Clear the screen"
    Display white.
    y ← Dot new.
    y center: 100@100 radius: 10.
    y display.
    200 to: 600 by: 100 do: [ : i |
            y center: i@i.
            y display ].
    y center: 300@200 radius: 20.
    y display.
    y radius: 30.
    y display.
    "Use a cascading expression for the rest"
    y radius: 40; display ; radius: 50 ; display ; radius: 60 ; display ; radius: 70 ; display.
    "Now write a short message on the transcript to remind user
    to refresh the screen"
    Transcript refresh.
    Transcript show: 'Press any mouse button to refresh display' ; cr.
    Sensor waitButton.
    "Refresh screen"
    ScheduledControllers restore

exampleThree

    "Display dots following the mouse. Hit blue (right) button to stop.
    Hit red button to leave a dot.
    To execute, select next line and do it:
        Dot exampleThree
    "
    | y r stop |
    "Clear the screen"
    Display white.
    "Get a new dot"
    y ← Dot new.
    y radius: 20.
    "Change the cursor to cross hairs"
    Cursor crossHair show.
    stop ← false.
    [ stop ] whileFalse: [
        Sensor redButtonPressed ifTrue: [
            y center: Sensor mousePoint.
            y display ].
        Sensor blueButtonPressed ifTrue: [
            stop ← true ]
    ].
"Return to normal cursor"
Cursor normal show.
"Refresh screen"
ScheduledControllers restore
```

exampleTwo

```
"Display some random dots.
To execute, select next line and do it:
    Dot exampleTwo
"
| y r |
"Clear the screen"
Display white.
"Get a new dot"
y ← Dot new.
"Get a random number Stream"
r ← Random new.
"Draw 20 random dots"
20 timesRepeat: [
    y center: (r next * 800) @ (r next * 600).
    y radius: (r next * 30) asInteger.
    y display ].
"Now write a short message on the transcript to remind user
to refresh the screen"
Transcript refresh.
Transcript show: 'Press any mouse button to refresh display' ; cr.
Sensor waitButton.
"Refresh screen"
ScheduledControllers restore
```

Flower

class name	**Flower**
superclass	**Dot**
instance variable names	**petalLength**
class variable names	*none*
pool dictionaries	*none*
category	**Graphics-Bubbles**

comment

Instance Protocols For: accessing

petalLength

```
    "Return my petal length"
    ↑ petalLength
```

petalLength: newLength

```
    "Set the my petal length to newLength"
    petalLength ← newLength
```

Instance Protocols For: displaying

display

```
    "Display myself on the screen.
    This method relies on the display method of the superclass Dot
    to draw the dot in the center, and then it adds the petals only"
    | petal c |
    "Display the petals. Use the Circle object to draw little circles.
    Get a new circle, and set the for (the width) to a dot of size 1.
    The radius of the circle is the petal length."
    petal ← Circle new.
    petal form: (Form dotOfSize: 1).
    petal radius: petalLength.
    "Draw six petals with this loop."
    0 to: 300 by: 60 do: [ : angle |
        "Calculate the center of the petal"
        c ← angle degreesToRadians cos @ angle degreesToRadians sin.
        c ← c * radius + center.
        "Set the center of the petal"
        petal center: c.
        "and display it."
        petal display ].
    "Display the dot "
    super display
```

```
"- - - - - - - - - - - - - - - - - - -"

Flower class
instanceVariableNames: ''
```

Class Protocols For: examples

example

```
"Paint two flowers. To execute this example select
the next line and do it.
    Flower example
"
| aFlower |
"clear the display"
Display white.
"Get a new flower"
aFlower ← Flower new.
"Size the flower and ask it to display itself"
aFlower center: 200@200 radius: 50.
aFlower petalLength: 20.
aFlower display.
aFlower center: 400@250 radius: 50.
aFlower petalLength: 60.
aFlower display.
"Remind refresh display after this little graphics"
Transcript refresh; show: 'Press any mouse button to refresh display'; cr.
Sensor waitButton.
ScheduledControllers restore
```