3

RUNNING THE EXAMPLE

CONTINUING THE FIRST EXAMPLE

One must learn by doing the thing;
For though you think you know it,
You have no certainty until you try.
SOPHOCLES, *Trachiniae*

The Smalltalk programming environment tries to provide every tool you want for finding, viewing, writing, and running Smalltalk methods. The system can tell that a particular piece of text is a method by the window in which it is typed. Thus there is no need for special punctuation marking the beginning or end of a Smalltalk method. Every time you deal with a whole method, it is inside a window that expects a method, such as area E of the browser.

The method moveDisk:to: includes a couple of new things.

```
moveDisk: fromPin to: toPin
   "Move a disk from a pin to another pin. Print the results in the
      transcript window"
   Transcript cr.
   Transcript show: (fromPin printString,' - > ', toPin printString).
```

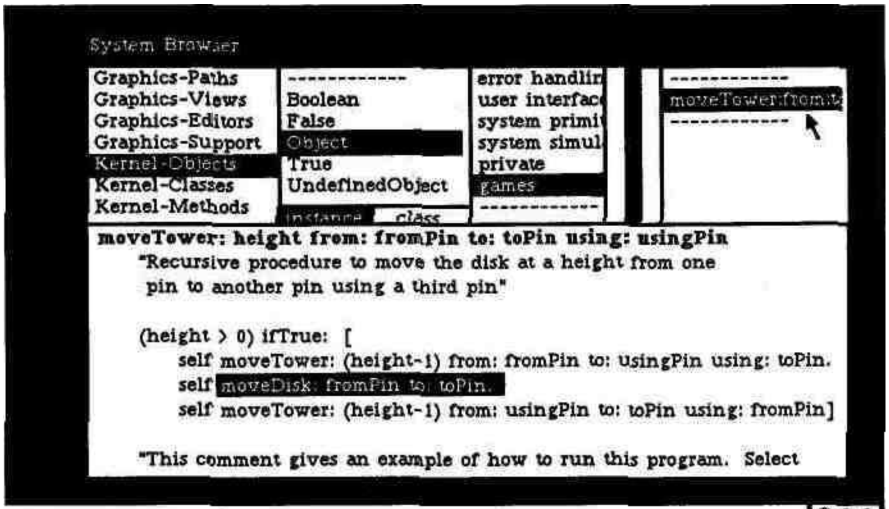
The System Transcript is a window on the screen. It behaves like a traditional character-oriented terminal. The object that represents the transcript is held in the global variable, Transcript. The message cr tells the transcipt to append a carriage return. The message show: takes a string as an argument and appends it to the transcript. It also redisplays the text in the transcript window, so you can see it,

Each comma in the expression (fromPin printString, ' -> ', toPin printString) is not a piece of punctuation, but an operator like +. The operator (selector) comma means "concatenate two strings." In this case, fromPin printString returns a string, and the comma concatenates the literal string' -> ' onto the end of it. The result oftoPin printString is then concatenated to the end of that. When we work with arrays in a later chapter, you will see that an array also understands the message comma. It returns a new array consisting of itself concatenated with the argument.

RUNNING A PROGRAM

If you remember from our last episode, we defined a new method (procedure) whose selector (name) is moveTower:from:to:using:. Now we wffltype in moveDisk:to: and run the program.

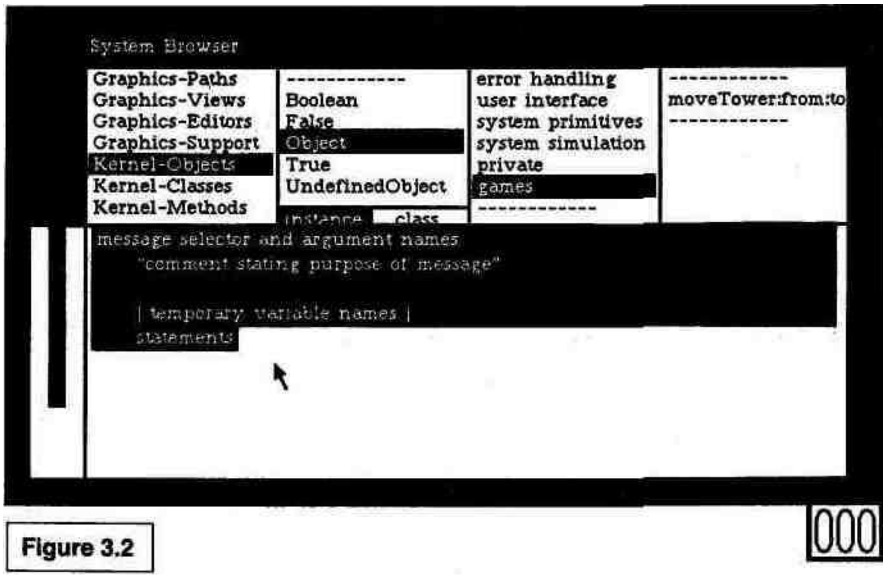
We need to clear a space in the browser, in which to put our new method. Go to area D and find moveTower:from:to:using:. It is probably selected (reverse video). See Figure 3.1.



```
moveTower: height from: fromPin to: toPin using: usingPin
    "Recursive procedure to move the disk at a height from one
     pin to another pin using a third pin"

    (height > 0) ifTrue: [
        self moveTower: (height-1) from: fromPin to: usingPin using: toPin.
        self moveDisk: fromPin to: toPin.
        self moveTower: (height-1) from: usingPin to: toPin using: fromPin]

    "This comment gives an example of how to run this program.  Select
```

Figure 3.1

000



```
message selector and argument names
    "comment stating purpose of message"

    | temporary variable names |
    statements
```

Figure 3.2

We don't want any method to be selected, so click once on moveTower:from:to:using: to "deselect" it. Now select all the text in area E (use the left button; press down at the beginning of the text, move to the end and release). See Figure 3.2.

Type in the code for moveDisk:to:

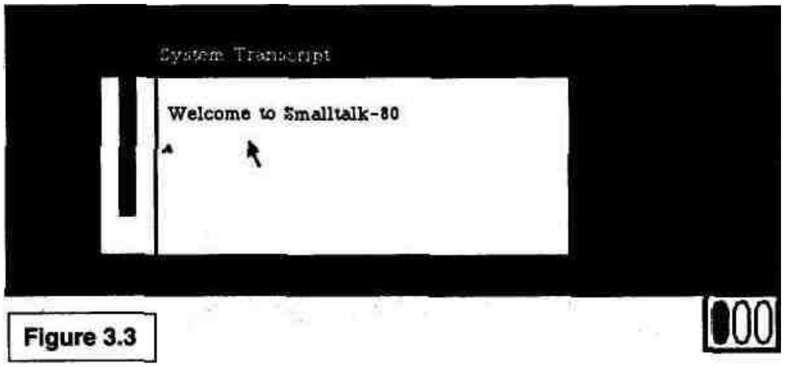
```
moveDisk: fromPin to: toPin
  "Move a disk from a pin to another pin. Print the results in the
    transcript window"
  Transcript cr.
  Transcript show: (fromPin printString,' -> ', toPin printString).
```

Once again choose **accept** by holding down the middle button, moving to **accept,** and releasing. (We call this "accepting a method.") The procedure name (message selector) moveDisk:to: should appear in area D of the browser. If anything else happens, such as an error correction window appearing on the screen, consult the section on "Troubleshooting When You **accept** a Method" in Chapter 2.

Let's try the Tower of Hanoi using three disks. Move the cursor to the window labeled System Transcript (probably in the upper left corner of the display) and click the left button ("enter the window") (see Figure 3.3).

When the window wakes up, point to the end of the text, click once, and type:

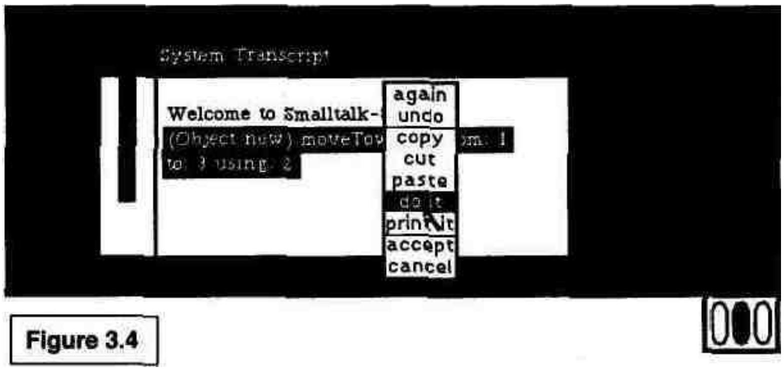
```
(Object new) moveTower: 3 from: 1
to: 3 using: 2
```



```
Welcome to Smalltalk-80
```

Figure 3.3

Select both lines of text and choose **do it** from the middle-button pop-up menu (see Figure 3.4).



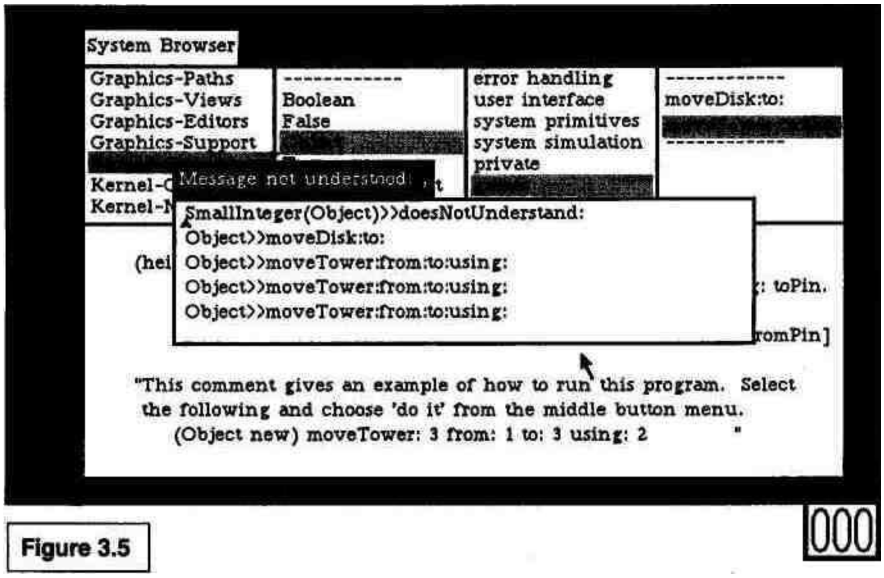
```
Welcome to Smalltalk-
(Object new) moveTov          om: 1
to: 3 using: 2
```

Figure 3.4

do it, as you might expect, tells Smalltalk to execute what you have selected. The System Transcript shows program output, just as a traditional character-oriented terminal does in other programming systems. This sequence of moves shouldscroll by:

```
1 ->3
1 ->2
3 ->2
1 ->3
2 ->1
2 ->3
1 ->3
```

If you missed what was printed, you can scroll the transcript back to see lines above the window. Just move the cursor to the left edge of the scroll bar until the cursor shows a down-pointing arrow. When you



```
SmallInteger(Object)>>doesNotUnderstand:
Object>>moveDisk:to:
Object>>moveTower:from:to:using:
Object>>moveTower:from:to:using:
Object>>moveTower:from:to:using:
```

```
(hei                                                    : toPin.
                                                        romPin]
"This comment gives an example of how to run this program.  Select
 the following and choose 'do it' from the middle button menu.
     (Object new) moveTower: 3 from: 1 to: 3 using: 2          "
```

Figure 3.5

click, the line at the top of the window will come to where the cursor is.

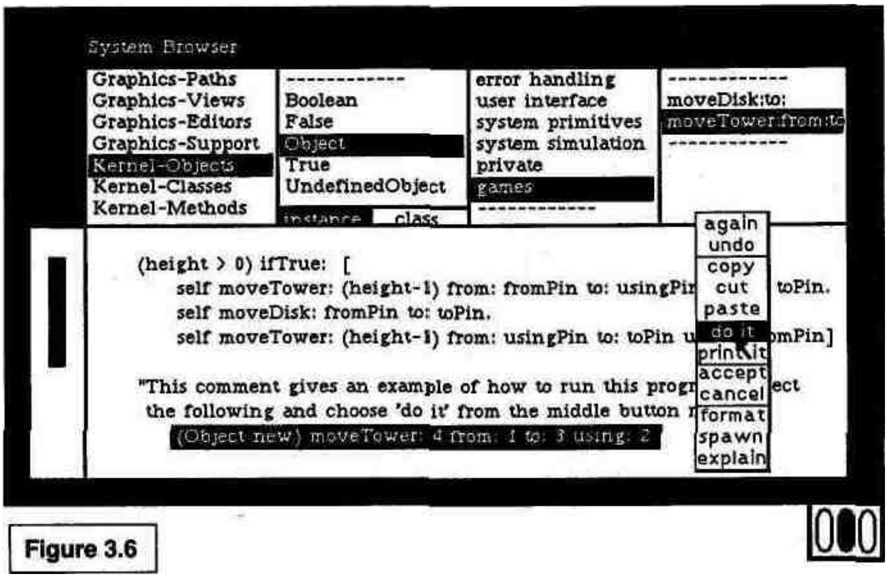
If all goes well, you will see the numbers in the transcript. If not, an error window like the one in Figure 3.5 probably appeared on the screen.

If the error window appears, please read the troubleshooting section that appears after this section.

Let's run the program again with different arguments. Rather than typing the line over again, use a comment you typed earlier. Enter the browser (click once after moving to it) and click on moveTower:from:to:using: in area D. At the bottom of the code, inside a comment, is an example of how to run the program. We can run it again with another height just by editing the comment to make the first argument (the height) be 4 instead of 3. Then select the example and say **do it** with the middle button. (Well, as most Smalltalk systems don't recognize human speech, you may have to use your fingers.) See Figure 3.6.

Smalltalk programmers traditionally show an example of how to use a given method in a comment inside that method. This means the scrap of code to run an example of the method is written only once, thereby avoiding the keystrokes of those long compound procedure names.

Before we go any further, please accept our congratulations! You have now written your first Smalltalk program. You could stop now and begin waxing eloquently about the strengths and weaknesses of



```
(height > 0) ifTrue: [
     self moveTower: (height-1) from: fromPin to: usingPi        toPin.
     self moveDisk: fromPin to: toPin.
     self moveTower: (height-1) from: usingPin to: toPin u        mPin]

"This comment gives an example of how to run this progr        ect
 the following and choose 'do it' from the middle button r
     (Object new) moveTower: 4 from: 1 to: 3 using: 2
```

Figure 3.6

Smalltalk, but if you hang on a bit longer we promise there is more fun. After all, we haven't yet told you how to declare new data types.

Notice that ^ve never declared the type of the arguments to our procedure, moveTowerrfrom:to:using:. In this instance, Smalltalk is much more like LISP than Pascal, because Smalltalk variables can be any type. Without making changes to the procedures, we can actually run the program with strings as the names of the poles instead of integers! Furthermore, the height can be a floating point number. Try the example again with new pole names:

```
(Object new) moveTower: 3.0 from: 'North'
to: 'South' using: 'Telephone'
```

When you select this and choose **do it,** the transcript will show:

```
'North'  -> 'South'
'North'  - > Telephone'
'South'  - > Telephone'
'North'  - > 'South'
Telephone'  - > 'North'
Telephone'  - > 'South'
'North'  - > 'South'
```

Smalltalk allows the same program to work on many different types of objects as long as they understand the same messages (operators and procedure names). For example, height need only understand > and -, and the pole numbers must only understand printString. Since every object understands printString, the method moveTower:from:to:using: is not very selective about its arguments.

TROUBLESHOOTING RUNTIME ERRORS

Let's review what to do if an error window appears on the screen while you are running your program. If you have any trouble in later chapters, you can refer to this section to help you find the problem. If this section becomes tedious, just skip it and go on to the next example.

Figure 3.5 in the section above shows a typical error window. The title tells you what happened. In this case it says. Message not understood: ,. The object that received the message comma did not know what to do in response to that message. Typically, this means that the receiving object is not the object you expected it to be when you wrote the code. The list inside the window shows the stack of nested procedure calls at the time of the error. It tells what methods were running or were waiting for an answer from methods they themselves called. In this example, the nesting is:

```
SmallInteger(Object)»doesNotUnderstand:
Object»moveDisk:to:
Object»moveTower:from:to:using:
Object»moveTower:from:to:using:
Obj'ect»moveTower:from:to:using:
```

The title and the top line tell us that a SmallInteger did not understand the message comma. The integer was sent the message comma in the method for moveDisk:to: in class Object. This is the code we just wrote, so it is a prime suspect for errors. Close the error window by placing the cursor in the window and choosing **close** from the right-button pop-up menu. Enter the browser and look carefully at the code for moveDisk:to:, especially where the message comma is sent. (In this hypothetical example, the user left out one of the printString messages in the last line of moveDisk:to:.)

For the example programs in this book, the errors will come from code that you typed in. Concentrate your search on the code you entered and are trying for the first time. Later, when you are modifying programs on your own and get an error, you can open the error window by choosing **debug** from the middle-button menu. The window expands into a complete debugger, which is explained in detail in Chapters 18 and 19 of the User's Guide.

THE SECOND SMALLTALK EXAMPLE

If a program is useful,
it will have to be changed.
ANONYMOUS, SICPLAN Notices Vol. 2, No. 2

So far we have called procedures and demonstrated output. Next we will modify our program to allow the user to type in the number of disks. Then we will learn how to save the program to a file on the disk. The method hanoi below asks the user for the number of disks and then calls moveTowerfrom:to:using:.

```
hanoi
   "Tower of Hanoi program. Asks user for height of stack of disks"

   | height aString |
   aString <- FillInTheBlank request: 'Please type the number of
disks in the tower, and <cr>'.
   height <- aString asNumber,
   Transcript cr.
   Transcript show: (Tower of Hanoi for:', height printString).
   self moveTower: height from: 1 to: 3 using: 2.

   "   (Object new) hanoi.   "
```

The third line of the method declares two local variables, height and aString. Although you do not need to declare the type of variables in Smalltalk, you do declare the names. Names enclosed in vertical bars at the beginning of a method are local to that method.

The next statement puts a fill-in-the-blank window on the screen and stores what the user typed into aString. Strings respond to the message asNumber which converts a string of digits (0-9) to a number. The result is stored in height. From there, we send a carriage return to the transcript, write some text there, and start the game running.

Left-arrow is the assignment operator; you can pronounce it as "gets." In one of the lines above, height gets aString asNumber.

When you are exploring the Smalltalk system, you will often be reading code. Reading code is easy if you can do two things: identify the objects, and understand the order in which messages are sent. All words that begin with a letter and don't end with a colon are objects, except those that immediately follow another object. In the expression aString asNumber, aString is an object and asNumber is a message.

Numbers, literal strings ('ABC'), literal characters ($A), and the results ofall message sends are also objects. In a series ofwords without colons such as height printString size, the first word is an object, and the rest are message names. In (height printString) size, the token size is still a message name, since the expression in parentheses evaluates to an object. See if you can underline all the objects in the method hanoi. (Look again at the example in Chapter 2.)

The order in which messages are sent is a little more complicated. As you might expect, parentheses are used to show what to do first. We were amazed to find that parentheses are not required in any of the first three methods we wrote. We will have to concoct an example in which they are required. Let's combine two of the statements in hanoi into one.

```
    height <- (FillInTheBSank request: ''Please type the number of
disks in the tower, and <cr>') asNumber.
```

If the parentheses were not present, we would be sending the message asNumber to the literal string (the stuff in single quotes) instead of sending it to the result of the request: message. When you write code you can include as many parentheses as you want, but you may read code that has a minimum of parentheses, so it helps to know the order of evaluation.

Smalltalk has just two rules that tell which messages are sent before others. Messages without arguments (asNumber, cr, hanoi, etc.) are executed first. They take precedence over adjacent messages that are operators (+, -, *, =, >, comma, etc.). Thus aString size > 0 is the same as (aString size) > 0 and means send the message size to aString first. Operator messages are executed before adjacent messages whose names contain colons. Thus

```
Transcript show: (Tower of Hanoi for', height printString).
```

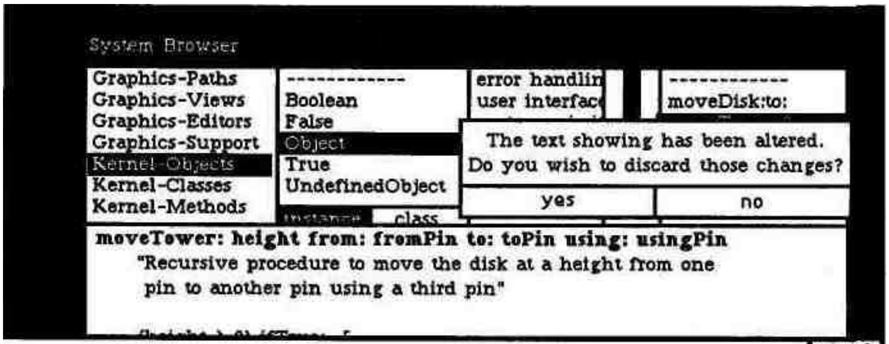
does not need parentheses to concatenate the two strings (Tower of Hanoi for ' and the result of height printString) before they are sent as the argument of the show: message. This is all explained in Section 5.2 of the User's Guide. In general, we have included more parentheses than are needed in order to make the code easier to read.

All spaces, tabs, and carriage returns in Smalltalk code are meaningless to the system. Packed methods compile just as well as beautiful, poetically indented ones. An exception is the separator that is needed between two names that are both alphabetic. aStringasNumber must have a space between aString and asNumber or it looks like a single name. Also, a period at the end of a statement must be followed by a separator to distinguish it from a decimal point. Keep in mind that, besides your machine, there is another computer that needs to understand your code. It is the human brain, and it greatly prefers nicely spaced and indented code.

INSTALLING THE hanoi METHOD

To clear area E of the browser, you first have to choose **cancel** from the middle-button pop-up menu (in area E). **cancel** means that we don't want to keep the changes we made to the arguments of moveTower:from:to:using: in the comment. You may have noticed that the middle and right buttons are only used for pop-up menus. The right-button menu holds commands that act on an entire window, and the middle-button menu commands are specific to the area of the window that the cursor is in. The left button never has a pop-up menu on it, and is reserved for pointing and clicking on fixed menus.

Now click on moveTower:from:to:using: in area D to deselect it. If you forgot to say **cancel** before choosing another method, a window will ask you if you want to discard the old changes. To confirm, move the cursor into the **yes** box and click the left button (see Figure 3.7).*



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

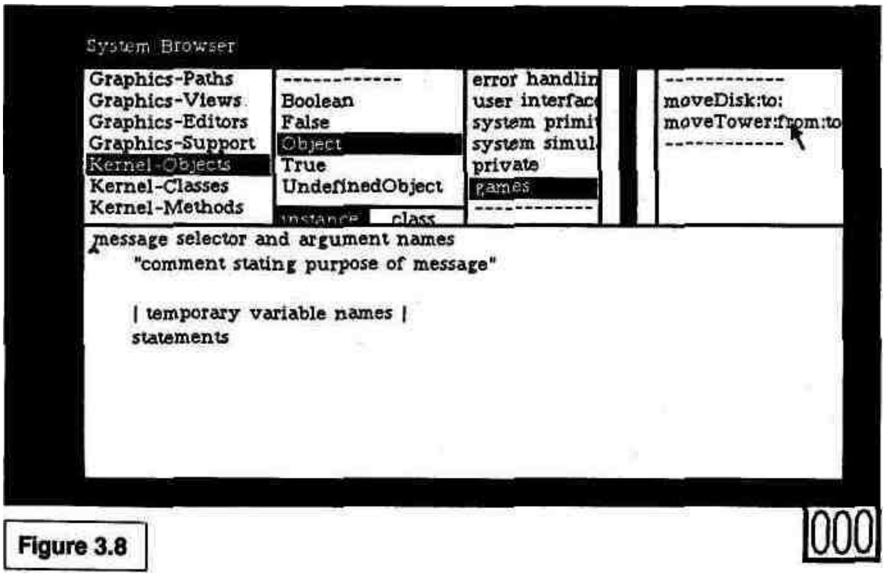
```
moveTower: height from: fromPin to: toPin using: usingPin
    "Recursive procedure to move the disk at a height from one
    pin to another pin using a third pin"
```

Figure 3.7

The browser should now look like it does in Figure 3.8. The text in area E is the default template for building a new method.

Select it all and replace it with the method below. The vertical bar character should be on your keyboard. If it is not, look for a broken vertical bar !. If your keyboard does not have a left-arrow key <—, use underscore _ instead. (Some keyboards have a <— key for moving the

* If you are running Apple's Level 0 Smalltalk system for the Macintosh 512K, your system may be missing two methods that hanoi uses. If so, please turn to Appendix 3, and follow the directions there to add the missing code to your system. All other readers may ignore Appendix 3.



```
message selector and argument names
    "comment stating purpose of message"

    | temporary variable names |
    statements
```

Figure 3.8

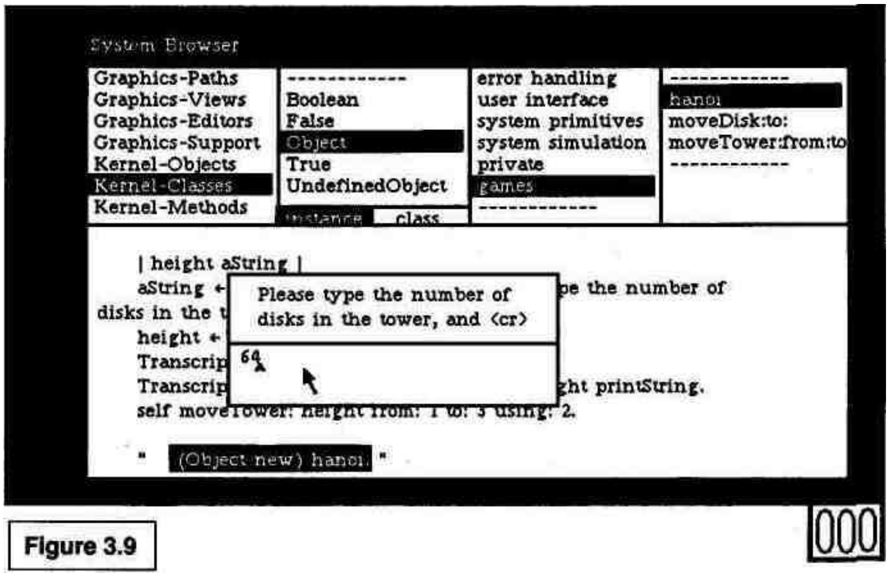
cursor. That key will probably not do the right thing. Ifthe underscore key is the correct key to use on your system, Smalltalk will change it to a left arrow on the screen.)

```
hanoi
  "Tower of Hanoi program. Asks user for height of stack of disks"

  | height aString |
  aString <- FillInTheBlank request: 'Please type the number of
disks in the tower, and <cr>'.
  height <- aString asNumber.
  Transcript cr.
  Transcript show: Tower of Hanoi for:', height printString.
  self moveTower: height from: 1 to: 3 using: 2.

  "  (Object new) hanoi.  "
```

Be sure to say **accept** (middle-button pop-up menu). If anything else happens, such as an error correction window appearing on the screen, consult the section on "Troubleshooting When You **accept** a Method" in Chapter 2.

We run this program by selecting the comment (Object new) hanoi (select inside the double-quote characters) and then choosing **do it** from the middle-button pop-up menu. A small window will appear and ask for the number of disks. Type any number and then hit the return key (see Figure 3.9).

It is interesting that few Smalltalk programmers use input param-



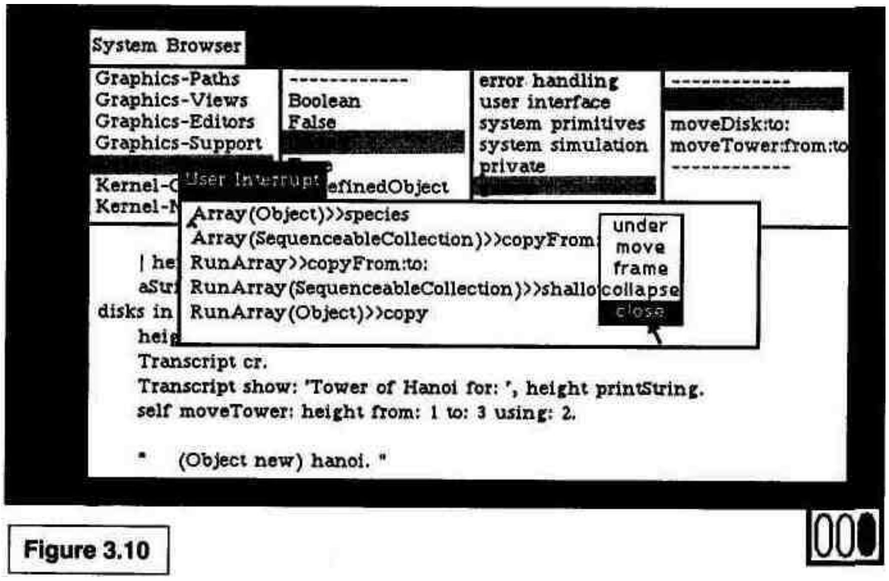
```
| height aString |
aString ←                                    pe the number of
disks in the t
height ←
Transcrip
Transcrip                                    ght printString.
self moveTower: height from: 1 to: 3 using: 2.

"   (Object new) hanoi. "
```

```
Please type the number of
disks in the tower, and <cr>

64
```

Figure 3.9

eters—they just edit a line that calls the program to include new values. Compiling and linking are so fast that there is rarely a reason to use input to get different values for parameters.

By the way, if you happened to type the number 64, and would like to change your mind to avoid bringing the universe to an end, just type *control C* (on a Macintosh it's *Command period).* To discard that semi-infinite process, close the error window (using the right-button pop-up menu to choose **close)** (see Figure 3.10). You can then make



```
Array(Object)>>species
Array(SequenceableCollection)>>copyFrom:
RunArray>>copyFrom:to:
RunArray(SequenceableCollection)>>shallo
RunArray(Object)>>copy
```

```
| he
aStr
disks in
hei
Transcript cr.
Transcript show: 'Tower of Hanoi for: ', height printString.
self moveTower: height from: 1 to: 3 using: 2.

"   (Object new) hanoi. "
```

Figure 3.10

sure (Object new) hanoi is still selected, choose **do it,** and enter a less catastrophic number of disks.

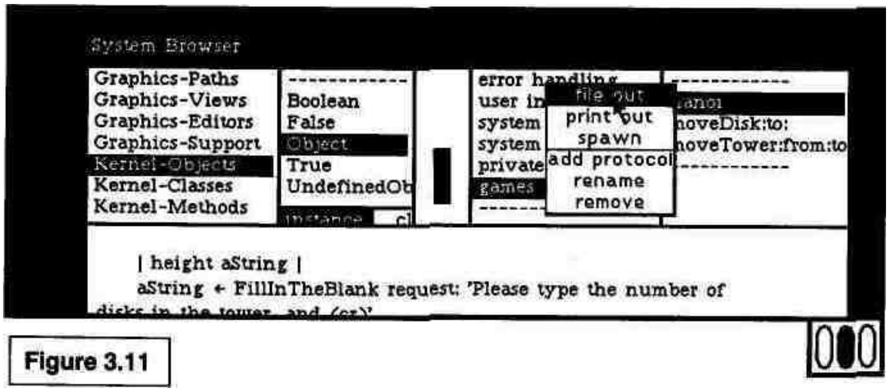
Congratulations are called for again. You have now written a Smalltalk program that includes input as well as output.

SO YOU DON'T WANT TO TYPE THIS ALL IN AGAIN

The first rule of intelligent tinkering
is to save all the parts.
PAUL EHRLICH (environmentalist)

As with any interactive system, it is important to save your work. We would like to write out a file containing the three methods we have defined so far. Go into area C in the browser and hold down the middle button. Choose **file out** from the menu (see Figure 3.11). (If the menu says only **add protocol,** you need to move out of the menu, release the button, and select **games** again by clicking it with the left button.) The system will name the file Objects-games.st and write it on the disk.

If you are tired, this is a good time for a break. When you leave the Smalltalk system, you have two choices for saving your current state. If you want to start your next session exactly where you left off this time, you can "make a snapshot" and save the *entire* system on the disk. If you don't mind going back to the system from which you started this session, you can quit without saving anything. Since we just wrote out our program in a separate file, Objects-games.st, and we want to get experience bringing that file back into the system, let's quit without saving anything.



```
| height aString |
aString ← FillInTheBlank request: 'Please type the number of
disks in the tower, and <cr>'
```

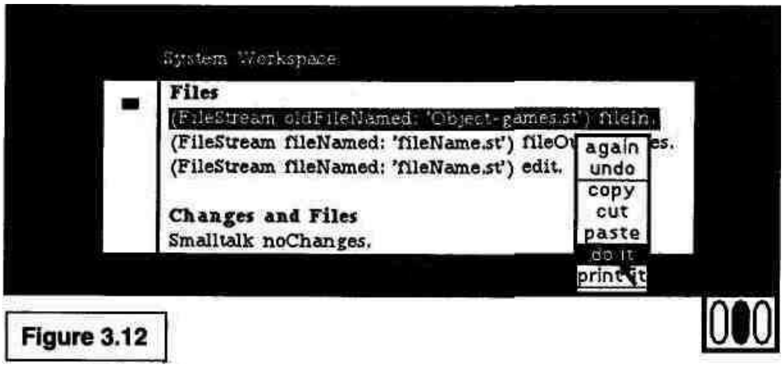
Figure 3.11

To exit Smalltalk, move the cursor into the gray background that is not in any window. Hold down the middle button, and choose **quit** from the menu. Another small menu will appear with three choices: **Save, then quit; Quit, without saving;** and **Continue.** Choose **Quit, without saving.** You will leave Smalltalk and enter your machine's operating system.

After getting back into Smalltalk, let's bring in the programs we wrote by reading the file we created. Move the cursor to the System Workspace window in the upper right corner of the screen.* We have not used this window before, but it contains many useful Smalltalk expressions that can be edited and executed. Enter the System Workspace window and scroll to the section called Files. (In License 1 systems the section is called Changes and Files.) You may need to get your sea legs again on the scroll bar. Side-to-side movement changes the cursor, and the up-arrow moves the line beside it to the top. Modify a line to say:

```
(FileStream oldFileNamed: 'Object-games.st') fiteln.
```

then select the whole line and choose **do it** from the middle-button menu (see Figure 3.12).


System Workspace

```
Files
(FileStream oldFileNamed: 'Object-games.st') fileIn.
(FileStream fileNamed: 'fileName.st') fileOu     es.
(FileStream fileNamed: 'fileName.st') edit.

Changes and Files
Smalltalk noChanges.
```

Figure 3.12

If all goes well, Smalltalk should put the following in the transcript window:

```
Filing in from:
Object-games.st
ObjecKgames
```

* If no window on your screen is labeled System Workspace, move the cursor to the gray area and hold down the middle button. Choose **system workspace** from the menu that appears. When the cursor changes to a corner shape, press and hold the button and move the cursor to where you want the other corner of the new window. On the Mac 512K there is no System Workspace. Just type the line (FileStream oidFileNamed: • Object-games.st') filein into the transcript, select it, and do It.

Now you can enter the browser, choose the category **Kernel-Objects,** choose Object, and see your methods waiting there for you.

The code in parentheses returns an object that is a stream on a 6!e. The message filein causes the stream to invoke the compiler and to parse the contents of the Ble in a special way. The Smalltalk code in the file is in "Smalltalk-80 code file format." Not only are the methods you accepted there, but they are exactly where you put them in the browser. (Hackers who absolutely *must* know about code file format can read Glenn Krasner's article about it in Chapter 3 of *Smalltalk-80: Bits of History, Words of Advice.)*