Mops

Mike's Object-oriented Programming System

Version 2.6

Part I

Introduction and Tutorial

Mops is an object-oriented programming system, derived from the Neon language developed by Charles Duff and sold by Kriya, Inc. Kriya have discontinued support for Neon, and have released all the source code into the public domain, retaining only the ownership of the name Neon.

Mops implemented by: Michael Hore

Able assistance from: Doug Hoffman
Greg Haverkamp
Xan Gregg

Documentation updated: Version 2.6, June 1995

Documentation formatted by: Craig Treleaven

Printing this document

This document is in Microsoft Word Version 5.1 format and uses the fonts Times, Courier, and Helvetica, only. It is formatted using the Laserwriter 7 driver for US Letter paper, portrait orientation, with fractional widths enabled. If you want to print any other way, you will probably need to repaginate and regenerate the table of contents and table of predefined classes and methods. See below.

Almost every paragraph in this document is formatted using a Word style. Formatting is consistent throughout and can be reformatted in moments this way.

Viewing on-line

Of course, you can read the whole manual on-screen. Word's Find… command can help to locate items of interest. One other technique is useful but not well known. Use the Outline View and click the "2" in the ruler at the top of the screen. Word will then show the Lessons and the sub-headings within the lessons. Whichever line is at the top of the window in outline view will become the line at the top of the window when you switch back to Normal View. By scrolling in Outline View, you can quickly find the section of interest and position the window for reading in Normal View.

Two-sided printing

As shipped, this document is formattted for 2-sided printing to save paper. If you haven't printed two-sided documents with your printer before, you might want to practise with the first few pages before sending the whole thing. On most printers, you need to use Word's option to print first the odd numbered pages (in the Print… dialog), reload the paper and then print the even numbered pages.

Single-sided printing

If you don't want to bother with two-sided printing, use the Document dialog and make the Gutter margin zero. If you adjust the Left and Right margins so the printable width is still 6.5 inches, the page breaks should stay in the same places. Blank pages may pop out here and there as all Lessons start on an odd-numbered page.

A4 Paper

If you select A4 paper in the Page Setup… dialog, the page breaks will change. Regenerate the table of contents, as below. As far as I can tell, the paragraph styles all do the right thing and ajust to the paper width. Well, all except one: the header on odd-numbered pages will extend a quarter inch into the margin because the tab stop is at 6.5 inches. Redefine the Header style to set it to 6.25, if you feel the need.

Table of Contents

Use the Table of Contents… dialog to collect headings from level 1 to level 2 for the Table of Contents. Figure captions have Heading 5 style, but I didn't see a reason to create a table of figures.

Introduction 1
Before you start............................1
Backup...2
Using an editor............................2
Mops—an object-oriented language 3
Private data..................................5
A threaded language.....................5
The Mops dictionary.....................6
Developing stand-alone applications 6
What your Mops system contains.6
The Bomb box.............................7
Internet and Web info for Mops. . .7

Lesson 1 **9**
How to start up Mops...................9
The Mops window......................10
The ENTER key.........................10

Lesson 2 **13**
More about the stack..................13
Arithmetic and the stack............15

Lesson 3 **17**
Stack notation.............................17
Mastering postfix notation..........19

Lesson 4 **21**
Mops and OOP...........................21
Methods and Inheritance............21
Objects and Messages................22

Lesson 5 **27**
Mapping class-object relationships27
Defining a class..........................27

Lesson 6 **31**
Objects and their messages.........31
Summary....................................33

Lesson 7 **35**
Modifying a Mops program........35

Lesson 8 **37**
Predefined classes—an introduction 37
Data structure classes.................38
Other predefined classes.............39

Lesson 9 **41**
Defining new Mops words.........41
The return stack..........................42
Named input parameters.............42
Local variables...........................43

Lesson 10 **45**
Additional math..........................45
Displaying text...........................45
Explicit stack manipulations.......46

Lesson 11 **49**
How Mops makes decisions.......49
Two alternatives.........................50
Truths, falsehoods, and comparisons 50
Nested decisions.........................52
Logical operators........................53
The CASE decision....................53

Lesson 12 **55**
Loops..55
Definite loops.............................55
Nested loops...............................57
LEAVE.......................................57
Indefinite loops...........................58
EXIT...59

Lesson 13 61
Mops' fixed-point arithmetic......61
Decimal, hex, and binary arithmetic 62
Signed and unsigned numbers....63
One last set of numbers—ASCII 64

Lesson 14 65
Global constants and values.......65

Lesson 15 67
Building a sine table...................67
How the sine table works...........70
Lines 1-5 70
Line 8 70
Line 11 70
Line 13 70
Line 15 71
Line 16 71
Lines 18-34 71
Lines 36-40 72
Lines 46-66 72
What happens on the stack.........73
Lines 71 - 76 75

Lesson 16 77
Building a turtle graphics program77
Line 3 79
Line 5 79
Lines 7 - 23 79
Lines 25 - 26 80
Lines 28 - 60 80
Lines 62 - 98 82
Lines 100 - 129 82
Lines 131 - 136 83
Experimenting with Turtle..........83

Lesson 17 85
Create a mini-Logo language.....85
Designing the language..............85
Implementing a Logo-like language 86

Lesson 18 89
Inside grDemo............................89
Views...90
Positioning views.......................90
Drawing views—the DRAW: method 93

Lesson 19 95
Windows....................................95
The grWind class........................96
dWind..96
Controls.....................................97
GrDemo scroll bars.....................98
Scroll bar actions........................98

Lesson 20 101
Menus......................................101
Running the program................102
In summary...............................103

Lesson 21 105
Installing an application...........105
Where to go from here.............109

Introduction

Mops is an object-oriented programming system, derived from the Neon language developed by Charles Duff and sold by Kriya, Inc. Kriya have discontinued support for Neon, and have released all the source code into the public domain, retaining only the ownership of the name Neon.

Mops is a complete re-implementation of Neon, with many additional enhancements. It is also in the public domain.

The name Mops could well be an acronym for "Mike's Object-oriented Programming System" but since I feel the computing world has enough acronyms already, I wouldn't want to be too dogmatic about this. Hence we spell Mops as Mops, not MOPS.

My hope is that over a period of time Mops users will, by sharing their developments, contribute to the ongoing Mops effort. As a one-man, very part time operation, I can't hope by myself to compete with all the commercial outfits producing gigantic, all-singing, all-dancing development systems for the Mac. I would be happy to concentrate on the low-level implementation of the Mops nucleus and basic system code.

In this introduction, we will introduce you to some basic concepts of Mops and its terminology. In the process, you'll see what a Mops program looks like, and how it differs from most other programming languages. We'll also take you on a guided tour of the Mops system.

Before you start

The first thing you should do, if you haven't done it already, is install the full Mops system. What you have so far is the application "Mops" itself, and a lot of source files. "Mops" is just the "nucleus" or "kernel" of the full system. To generate the full system, you have to use the nucleus to compile the source files. This is quite easy —just follow the step-by-step instructions below.

(The reason for this procedure, by the way, is to reduce the size of the total package as much as possible, by not having to include all the compiled binary files.)

Right, here we go. First start up the nucleus application "Mops". You should get a window appearing on the screen with no menus. The window will be showing a message telling you that it is only the nucleus, and that you should now compile the system by typing

```
load base <ENTER>
```

and look at the Readme.1st file, or the intro chapter of the manual. That's exactly what you're doing—so far so good!

This window is just a very rudimentary one (it can't be moved or anything). Its only purpose is to allow the full Mops system to be compiled. Once this is done, Mops will have a proper interface.

There will be a ">" prompt, and an "underscore" cursor, ready for you to type something. So now go ahead and type

```
load base
```

(followed by the ENTER key)

This will do the first stage of the compilation. **Note that Mops commands are terminated by <ENTER>, and that <ENTER> is not the same as <RETURN>!** Many Mac programs treat these two keys as equivalent, but Mops doesn't. Once you've used Mops for a while you'll come to appreciate the usefulness of this feature.

As lines you type to the Mops window are normally terminated by ENTER, we'll assume this from now on.

The compilation will take a couple of minutes. If no errors come up, you'll get a message saying that interim.dic has been saved, and the ">" prompt again.

interim.dic is the partial dictionary that has just been compiled. It will be in the "Mops *ƒ*" folder.

If something goes wrong in the next stages, you can get back to this point by double-clicking on interim.dic, without having to go back to the bare nucleus.

Now, to compile the rest of the dictionary, type

```
// sys.ld
```

and everything else should be compiled. It only takes a couple of minutes on most Macs. The names of the different files will appear on the screen as they are being compiled. If everything goes OK, at the end of the compilation you'll get the following message, which should be self-explanatory:

The Mops system is compiled. Now save the dictionary, by typing e.g.

```
save Mops.dic
```

then type bye to quit, and after that you'll be able to fire up the newly-compiled dictionary.

So do what it says—save the full dictionary by typing

```
save Mops.dic
```

and then quit by typing

```
bye
```

Then after that, you can then fire up the "proper" Mops system by double-clicking on Mops.dic.

Finally, if you use floating point, you can compile it by firing up Mops.dic and typing

```
// floating point
```

and when it's compiled, save with

```
save MopsFP.dic
```

After this, you should be able to fire up either the standard or floating point system by double-clicking on the dictionary image Mops.dic or MopsFP.dic.

Backup

We can't emphasize enough the importance of backup. Assuming you are using a hard disk, you should keep it regularly backed up. With programming, especially, system crashes are commonplace! But these shouldn't worry you, if your disk is regularly backed up.

It would also be good to keep an extra backup of the original Mops distribution, whether it was on disk or downloaded from an on-line service. Then, if you somehow destroy anything in the Mops system, you can easily get back to a working system.

Using an editor

Mops does not have a built-in editor, but works in close cooperation with Quick Edit, which was developed by Doug Hoffman especially for Mops, and is included here in the "Quick Edit ƒ" folder. If both Mops and QE are running, they communicate via Apple Events to perform a number of useful functions. From Mops, you can ask QE to open a particular source file, and this will also happen automatically when an error occurs. From QE, you can send text to be interpreted by Mops, or request that Mops locate a source file which QE then opens. QE also incorporates an on-line Mops glossary. So we do recommend you give QE a try.

To use the on-line glossary, just highlight a word, and choose Reference under the Special menu, or hit command-3. Also, in the Glossary window, start typing the name of the word you want to look up, and you'll be taken there. You'll find further instructions for using QE in the Readme file in the "Quick Edit ƒ" folder.

Mops—an object-oriented language

In Mops, much programming is done by sending messages to objects. A Mops object can be a simulation of any real-world object you're familiar with: a rectangle, a Macintosh window, an artist's canvas, a bank account. When a Mops program runs, a relatively small list of instructions route program execution through the framework of objects, and the objects come to life: a rectangle draws itself on the screen; a window appears; a moused-controlled brush paints on an artist's canvas; a bank account monitors income and payments.

Mops, itself, includes many pre-defined types of object. With this pre-existing framework, you can create complex objects as simply as typing two words. Let's do that right now, so you can get a taste of what Mops has in store for you. This is just a demonstration, not a lesson. So type along with us, and observe what happens without trying to remember each step.

Open the Mops ƒ folder. Locate the Mops.dic icon and double-click it. In a moment, the Mops window appears. We'll explain the window's contents in detail in Lesson 1 of tutorial, but for now, create a rectangle object—called "box"—in memory by typing:

```
rect box
```

(Remember to hit <ENTER> at the end.)

We need to tell box where on the screen it should appear, and how big it should be. The rectangle framework inside Mops wants these instructions in the form of screen coordinates for two opposite corners, the top left and bottom right. We'll choose 20, 10 for the top left, and 200, 100 for the bottom right. Put these figures into box's memory by typing the following line, making sure you observe the spacing between elements and the colon:

```
20 10 100 50 put: box
```

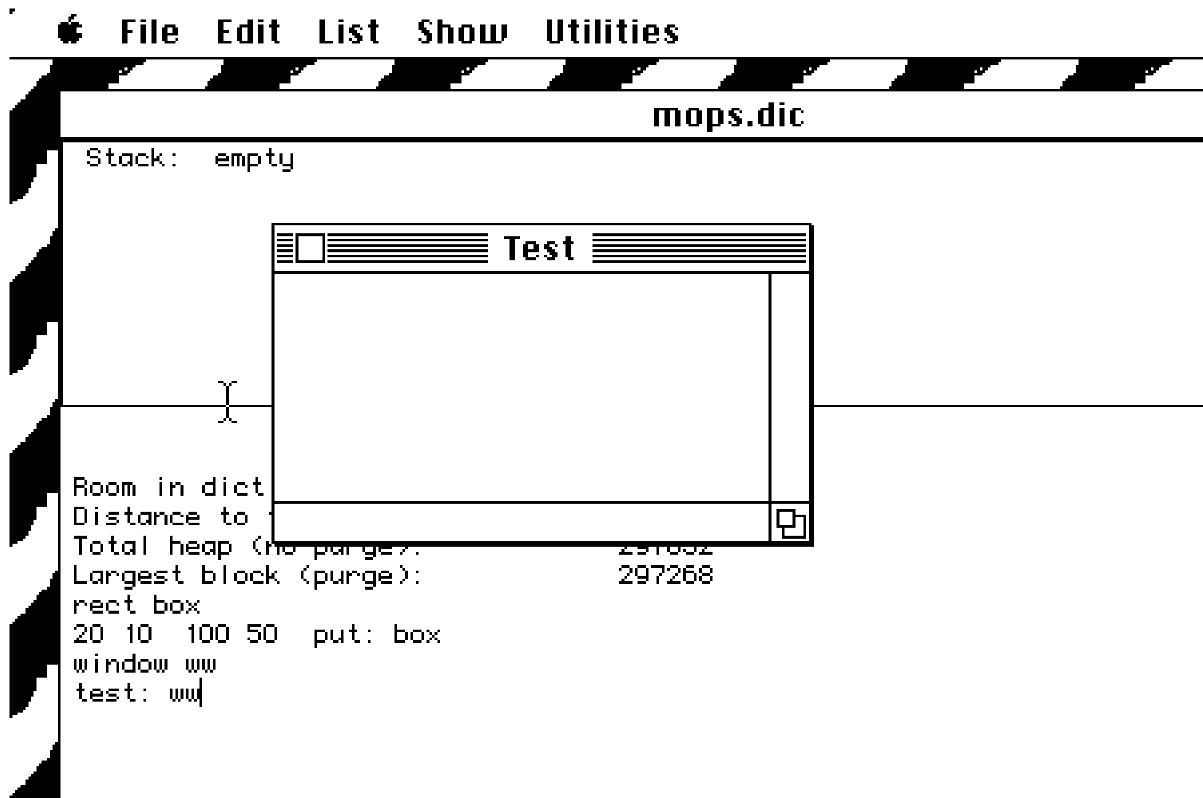
This line is called a message, which we just sent to box. Now we want to send messages to box so that it will draw itself on the screen. First, however, it will need a window to draw itself in. To set up a window object—named "ww"—in memory, type:

```
window ww
```

Macintosh windows need a lot of information before they can be placed on the screen, including the rectangular limits of the window, the title of the window, the type of window, whether it is to be visible, and whether it has a close box. Even then, the Macintosh Toolbox requires much more information, which Mops automatically supplies. Some of the Mops classes, including Window, have test or example methods that display an instance of that class, with typical values. To see the window you just created, type the following message:

```
test: ww
```

Your screen should now look something like this:

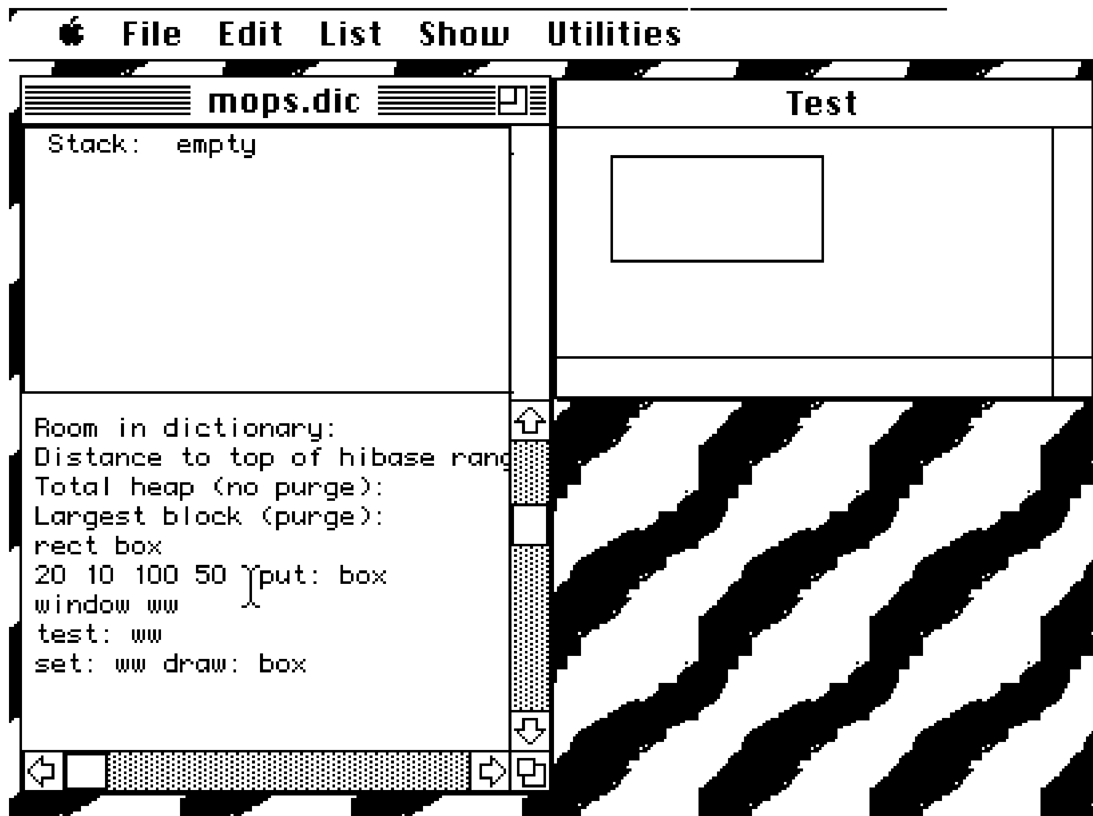


You will be able to resize and drag ww around the screen as for any Mac window. But if you type keys while ww is in front, nothing will happen—this is simply because we haven't told ww what to do with keys. So move ww out of the way, and click on the Mops window so you can type further commands. Resize the Mops window if necessary so that ww is still visible.

Now type:

```
set: ww  draw: box
```

The message "set: ww" tells the system that drawing is now to take place in ww (but without bringing it to the front). Box should now appear in ww, and your screen should look something like this (of course you might have put ww in a different place):



When you are finished experimenting, select Quit from the File menu, or type

```
bye
```

to quit Mops.

Private data

The values you assigned to "box" didn't just disappear after the rectangle appeared on the screen. A key element of Mops is that the parameters you plug into an object (like the opposite corner points of box) won't be disturbed, even if you were to create a new rectangle, called "square" and assign it entirely different values. You could create a hundred different rectangles with a hundred different names, locations, and sizes, and each rectangle object would hoard its own parameters as private data. Both an object and its private data stay in memory until called upon.

A threaded language

Even though a Mops statement like "draw: box" is rather simple, it is referencing several other statements (called definitions) that have been predefined for you in the Mops dictionary (which is contained in Mops.dic). When the statement "draw: box" is encountered, a chain reaction takes place inside the computer, as one definition reaches back for previous definitions, which, in turn, reach back for further definitions, until they reach Mops' primeval definitions at the very core—kernel—of the language. And yet, far from being cumbersome and slow, Mops is performing these self references in machine language—the fastest possible method of accessing information stored in memory. This "reaching through" memory for all connections to a given definition is why Mops is called a "threaded language."

The Mops dictionary

You can think of the "definitions" we've been discussing so far as if they were definitions in a dictionary of any language. The words are called Mops words, and you can always look up their definitions in the Mops Glossary, which is part of the Quick Edit editor.

The Mops language is a complete, interactive dictionary of predefined Mops words. When you double-click Mops.dic, the Mops dictionary is loaded into memory. When you write a Mops program, you essentially add definitions of your own Mops words to the kernel (the new definitions apply only to the particular program you're writing. For other programs you'll usually start with a fresh copy of the basic dictionary).

Unlike most procedural languages, which lock you into its vocabulary of step-by-step instructions, Mops lets you define one word to do the work of dozens of words. The ability to create new commands as you create your program is called extensibility, and you will find that many of the Mops words you define in your first few programs will be reusable in other programs. This will help you reduce development time for succeeding projects and free up time for further exploration of the vast richness of the Macintosh environment.

Mops comes with a large number of finished building blocks, called predefined classes. As we'll see in the Tutorial, a class defines a type of object. You have already used two of these classes: Class Rect and Class Window. You will be using these and other predefined classes while learning Mops and later in writing your own programs.

Developing stand-alone applications

Mops can be used to produce stand-alone double-clickable applications, whose users won't need to concern themselves with what language the application was written in. These users won't have or need access to the Mops dictionary and interpreter. Instructions for this procedure are given in the Tutorial.

What your Mops system contains

Once you have installed the Mops system as described above, you will find several folders containing Mops files. They are:

Mops ƒ	Essential files including: * The Mops kernel (Mops). * An image of a Mops dictionary with many of the predefined classes already loaded (Mops.dic).
Mops source	All the Mops source code. There are various other folders inside here, as follows:
Nuc source	Source files for the Mops nucleus (Mops itself). The listing and macros are in McAssembly format.
System source	Source code for the basic Mops classes and other support code.
Toolbox classes	Source code for Mops classes that interact with the Mac Toolbox.
Module source	Source code for the Mops modules.
Asm source	Source code for the Mops 68000 assembler.
Demo folder	Source files for demonstration programs used in the tutorial.

Most of the source code files in System, Toolbox, and Demo folders are provided not only for added documentation, but also if you want to recompile a modified version of Mops. A study of that code, along with the tutorial, will help you master the powers of Mops.

Mops.dic (or MopsFP.dic) is the predominant file you will be opening, just as you did earlier in this introduction. It contains the majority of the Mops words and predefined classes on which you will build programs.

The Bomb box

In the course of your experimenting with Mops, you will inevitably—and perhaps unknowingly—issue a command that the Mac doesn't understand. While many such errors will bring an error message to the screen (each is explained in Appendix A), other will summon the system error box, sometimes called the "Bomb Box" because of the icon it displays. Alternatively, if you have Macsbug installed, you might find that Macsbug has been entered and its display is on the screen. If this happens, you will possibly be able to warm start back to Mops by typing **`G 1E4 <return>`**. If you land straight back in Macsbug you'll have to reboot, by typing **`RS <return>`**.

If you reboot, this will completely reset the computer, erasing from memory whatever you were working on. It is a good idea, therefore, to save your work often—a guideline you should follow in all your computer work.

There will be other occasions when the computer will appear to "lock up." It will be unresponsive to your keyboard or mouse actions. In this situation you'll have to reset the machine to cause it to reboot. Many Mac models have a Reset button which will do this. Other models (e.g. IIsi) don't use a button, but use the command-control-power on key combination. If the computer appears frozen, press the Reset button or the key combination, and this will function the same as turning your Mac off and on again.

If you get the Bomb Box on the screen or lock up the computer, don't despair. It happens to the most experienced Macintosh programmers (it happens to me a lot!). Do your best to remember what you did to cause the problem and try to work out how to fix it.

Internet and Web info for Mops

If you have internet access, you'll find Mops discussion in the group comp.lang.forth.mac. Feel free to dive in there. Also note that I post a FAQ (frequently asked questions) list there every month or so. See if you can read the latest version of that before emailing me with a problem, since a lot of questions people ask me are already answered there. But feel free to email me with compliments at any time!!

There's a Mops web page at <URL:http://www.netaxs.com/~jayfar/mops.html>. This has links to the FAQ and the latest Mops version.

For ftp, the latest version should be at: taygeta.oc.nps.navy.mil, directory pub/Forth/Mops

Finally, if you do have to email me with a problem, remember to tell me what version you have! My email address is mikeh@zeta.org.au.

Now let's move on with the Tutorial, and I hope you find you enjoy Mops.

Lesson 1

How to start up Mops

First, a reminder—if you haven't done this already, you will need to compile up the full Mops system from the source. The instructions for doing this are in the Intro, and also in the Readme.1st file. The reason for this procedure is to reduce the size of the total package as much as possible, by not having to include all the compiled binary files. So if you haven't already done so, please read the instructions and do what they say.

To start Mops, if the "Mops ƒ" folder is not open, double-click its icon. Double-click the icon labeled Mops.dic. After a few seconds, the Mops window will appear on the screen, as in Figure I-1.

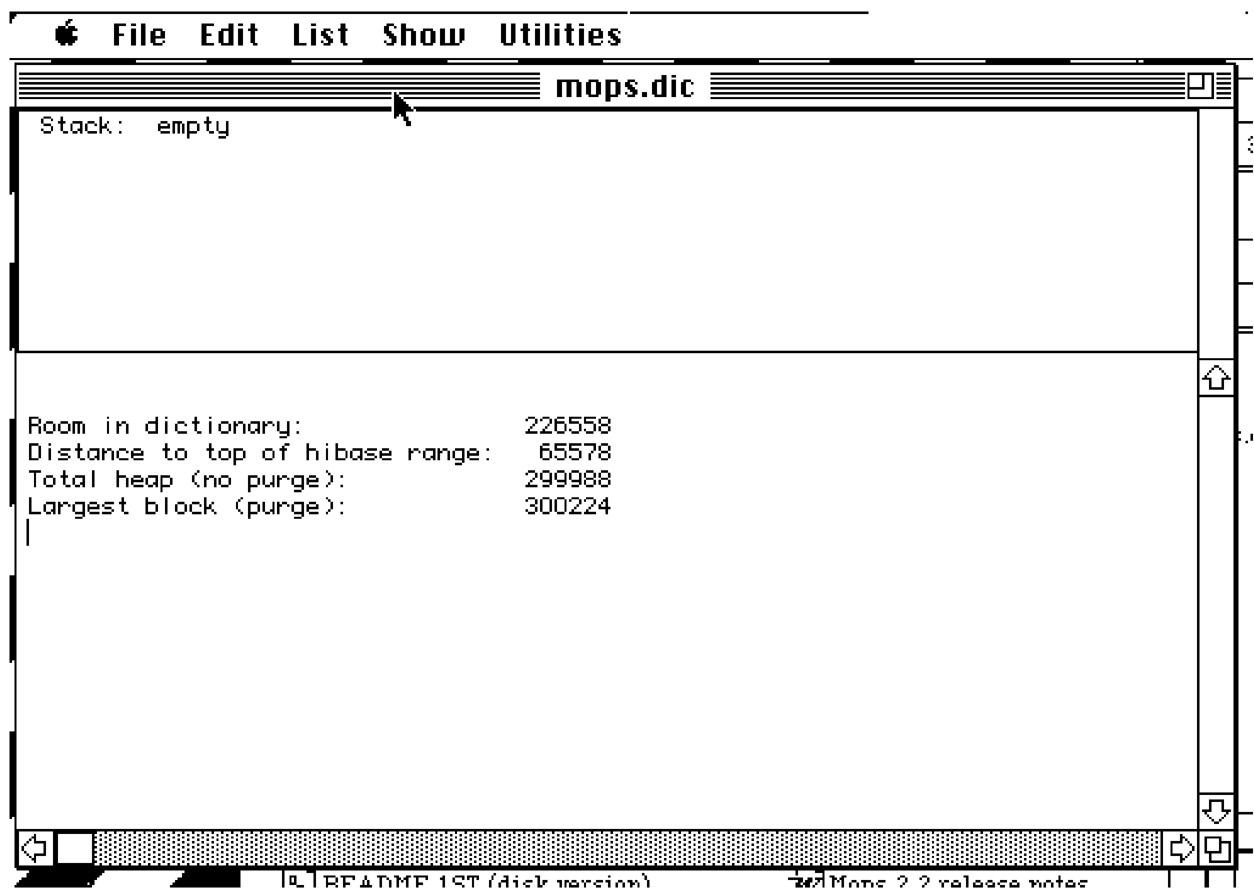


Figure I-1

As well as the Apple menu, you will see five menu titles. You won't be using all of the menus right away, but acquaint yourself now with the contents of each menu.

The Apple menu contains the usual items. The "about Mops" item causes the current version number of Mops to appear. The File menu is much like the File menu in other applications, but with a special item (Load), which you'll use later for loading text files containing your program code.

The Edit menu has the usual items—cut, copy, paste and clear. These can be used in the Mops window.

The remaining menus, List, Show and Utilities, contain many operations that will be useful in the writing and debugging of Mops programs. These operations are detailed in Part II of this manual.

The Mops window

For now, ignore the top part of the Mops window. In the lower part, you will see some information about how much memory is available in different areas—we'll be describing these values later.

Below these lines, you will see a flashing cursor, as you might see in a word processor or text editor. Type something, and you will see that this window is indeed a text editing window. You may type and edit text here, as in any text editor.

The ENTER key

There is one important difference however, which we have already seen in the Introduction, and that is with the use of the Enter key. In most text editors, Enter is treated the same as Return. In Mops, however, Enter causes Mops commands to be executed. We'll try this now.

We said earlier that Mops behaves like a dictionary. In other words, when you opened Mops.dic just now, the Mac automatically loaded the basic Mops vocabulary into its memory. Each time you type a word—any group of text characters—and press Enter, Mops searches through its dictionary for that word and carries out whatever instructions are associated with it. If the word you type is not in the current Mops dictionary, a message appears on the screen to advise you that Mops could not find the word. We'll try it in a moment.

If you're familiar with another computer language, note that in Mops, as in other Forths, we use the word "word" in a different way to normal computer terminology. A Mops "word" is any group of text characters, terminated by white space (a space, tab or carriage return). In C you might say "if(a>b)foo(bar);" but in Mops that would be one word, because it contained no white space. Characters which are punctuation or special characters in other languages can be part of a Mops word, since the only thing that terminates a word is a white space character.

Now we'll demonstrate the difference between the Return and Enter keys.

First type someone's name and press Return

```
michael <RETURN>
```

The line just sits there in the Mops window, exactly as if you'd typed it into a text editor. Nothing else happens. Now try it again, but this time use Enter:

```
michael <ENTER>

Error # -13 : undefined word
qwqwqw
     ^
Current object:  TW    Class:
Stack: empty
Return stack:  Depth 42
9511662  $9122EE ?NOTFOUND
9502140  $90FDBC NUMBER
9522950  $914F06 INTERPRET
```

A few things happened here. Since you used Enter, Mops tried to interpret "michael" as a command. The message coming back from Mops indicated that the name you typed in was undefined. This means that the name was not found in the dictionary—in that split second, Mops compared the name against over 1100 words in the Mops dictionary. Mops would also have beeped, indicating that something was wrong.

And finally, the flashing cursor appears below the message lines, indicating that Mops is waiting for you to type another command.

Now try typing the number 999, and then type Enter. This time no message will appear from Mops—it has simply accepted the number you typed in. Where is it? It has been placed on a stack. You'll be able to see it in the upper part of the Mops window, which is a display of the stack:

```
Stack:  depth 1
999
```

Now type the number 888 and type Enter. You will see that it is placed on top of the 999:

```
Stack:  depth 2
888
999
```

The stack depth is now two, since there are two numbers there.

You can now change the numeric base that Mops is using. Watch what happens when you change the base from decimal (the base that Mops starts in) to hexadecimal (hex for short). Type

hex <ENTER>

The stack display should change to

```
Stack:  depth 2
378
3E7
```

The numbers on the stack didn't actually change, but are now being displayed in hex instead of decimal. (If you are not very familiar with hex numbers, we give a fuller description in lesson 13.) To change back to decimal, simply type

decimal <ENTER>

Now try the same thing, typing HEX and DECIMAL in upper case. The results should be exactly the same. That's because Mops makes no distinction between upper and lower case letters when it comes to words in its dictionary. Internally, everything is converted to upper case.

Now type an undefined word again. The error message lines will again appear, but also notice that the stack becomes empty. This is another part of the Mops error handling—the stack is emptied on an error.

End of lesson 1

Lesson 2

More about the stack

Type the numbers 7, 4 and 1 on one line with a space between them, thus:

```
7 4 1  <ENTER>
```

After you press Enter, the stack display should now appear thus:

```
Stack:  depth 3
1
4
7
```

The space you typed between the three digits told Mops that you intended those three digits to be three different numbers. (You can equally well use two or more spaces, or the tab key.) If you had typed 741 instead, then the single number 741 would have been put on the stack. Understanding the way these numbers are entered and stored on the stack is of utmost importance at this stage of learning Mops.

The best way to demonstrate how a stack works is to summon the often-cited analogy of the springloaded pile of dishes you encounter in a cafeteria line. If you place one plate on the spring, it is obviously the first one that will come off the top. But if you place a second plate on top of the first, the weight of the second plate pushes the first one down one step, and the second plate is the one that will be picked up by the next customer in line. In other words, the last one put on the stack is the first one to be taken off the stack.

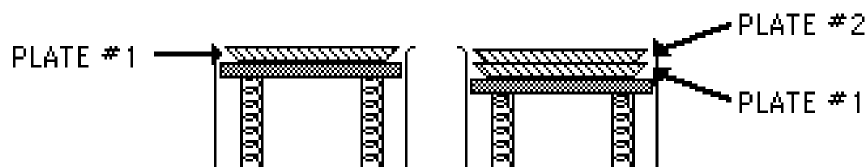


Figure 1-3.

You can see that this principle applies to the Mops stack, since the number 7, which you entered first, is at the bottom of the stack, while the number 1, which is the most recent entry, is at the top.

How can you remove numbers from the stack? One way is to use the Mops word that does the reverse of what you did when you entered a number—it takes a number off the stack, and "types" it in the lower part of the screen, at the flashing cursor position. That word is a simple period (.), called "dot." Type this now.

```
.   <ENTER>
1
```

What happened here was that the dot (type to screen) command pulled the 1 off the top of the stack and "typed" it. The stack display will show that two numbers are still on the stack. In other words, whenever you perform a dot operation on a number in the stack, the number is removed from the stack and "typed"—that is, displayed in the current window. If the Mops window is the current window, as it is here, any number "typed" will appear at the cursor position, just as if you had typed it at the keyboard.

To get the remaining numbers off the stack, you need to issue two more dot commands. Just as you could put two different numbers onto the stack by typing a space between them on one line, so you can issue multiple commands on one line, provided you put at least one space between each command. If you fail to put the required space there, Mops thinks that the string of characters is a single Mops word—perhaps a word that Mops cannot find in its dictionary. In Mops, one or more spaces or tabs are separators between words (which means, of course, that you can't have a Mops word with a space or tab in it).

To bring the cursor to the left margin, where it will be less confusing, simply press Return once. Remember that this won't try to execute anything—it will just start you on a new line in the Mops window. Now type two periods, with a space in between, and press Enter:

```
. .  <ENTER>
4 7
```

Mops has now printed the two remaining numbers in the order in which they came off the stack. Remember that the 7 was at the bottom of the stack; it was therefore the last number off the stack, and was displayed on the screen as the final item before the Mops prompt reappeared. Multiple dot commands, as you see, leave a trail of numbers off the top of the stack from left to right across the screen. And notice, too, that nothing remains in the stack when the last dot command has been executed.

Mops also has a word, .s (dot s), that displays a list of all numbers on the stack without removing them. This can be useful during the running of a program, since you may not want to stop the program to see the stack displayed at the top of the window. Also, there may be more items on the stack than can fit in the display. The stack can hold far more items than we have room to display at the top of the window, but if you use .s, all the stack items will be typed in the lower part of the window.

To see how it works, place the same three numbers on the stack (don't forget the spaces):

```
7 4 1   <ENTER>
```

And when you type .s, your screen will look similar to this:

```
.s  <ENTER>
Stack:  Depth 3
      1  $     1
      4  $     4
      7  $     7
```

The numbers to the left of the dollar sign are the decimal values, while the numbers to the right are the hexadecimal values. The dollar sign in this list is a hex indicator. In this case, it happens that the stack numbers in both bases are the same. Note, too, that the three numbers are still on the stack. The regular dot command displays and removes them while .s simply takes a snapshot of them.

Experiment with the operation of the stack by putting numbers on the stack, viewing them with the .s operation, and taking them off by printing them to the screen, either one at a time or in a series. As an added shortcut, you can use the CR command, which is short for "carriage return," after a dot command. If you type a "CR" as a command after one or more dot commands (remember to type a space between the last period and the CR), the Mops prompt returns to the left margin of the next line. For example:

```
1 10 100  <ENTER>
. . . CR  <ENTER>
100 10 1
```

If you accidentally issue one more dot command than you have entries on the stack, Mops will send you a message (along with the alert beep) that the stack is empty. Try it. No harm will occur.

The stack is also called the parameter stack, because a good many operations in Mops require that one or more values be present on the stack before the operation can be performed. These values, in computer jargon, are called parameters, and they are said to be passed, or handed to, an operation. Actually, the operation looks to the stack for the number(s) it needs, and pulls them off.

You saw a glimpse in the last section of how parameters work, when the parameter stack held values that were to be printed to the screen. The parameter stack, in other words, is a kind of holding box for values that many operations rely on. This concept will become clearer as we now discuss how Mops performs arithmetic.

Arithmetic and the stack

If you've ever used a Hewlett-Packard calculator, you are already familiar with keying in two values and then pressing the key that bears the symbol of the desired operation, such as + for addition or * for multiplication. You're actually utilizing a stack-type computer when you do this.

For those who have never touched an HP machine, the steps to add 2 and 7 go like this. First press the 2 key. The 2 is placed on the top of the stack. Then press the Enter key. This pushes the 2 one cell deeper into the HP calculator's stack, a place in the calculator's memory where values are temporarily held until they are needed for an operation. Then press the 7 key, which places the 7 on the top of the stack. Finally, press the + key, which reads each value from the stack (first the 7, then the 2) and adds them. The answer, 9, appears both in the display and on the top of the stack, ready for further operations, if desired.

Mops works very much the same way.

The step-by-step approach to add two numbers would be to put each number on the stack one at a time, and then press the + key as follows. (Remember to type Enter at the end of each line so that Mops will interpret these lines as commands. We'll assume this from now on, without saying so.)

```
7
2
+
.
9 cr
```

Let's follow what happened here. You should already understand how the stack counter increments each time you type a number and press Enter. In the third line, you type the operation, the + sign for addition. When you press Enter, the computer calculates the sum for you. Mops stores the sum on the stack—you will see one value, 9, on the stack at this point. Note, too, that the original numbers were taken off the stack by the addition operation. To type out the contents of the stack, and the result of your addition, you must issue the dot command.

Mops lets you perform all these manipulations in a simpler form—as a single line of instructions, with at least one space between each element. Here's how it looks:

```
7 2 + . cr
9
```

The line of instructions contains the same commands as the step-by-step method, but is much easier to type in. The only thing you miss along the way is a step-by-step view of the stack. But after all, it's the answer that should be important, not the momentary contents of the stack.

End of lesson 2

Lesson 3

Stack notation

Before we go further, you should become acquainted with a special notation that tells someone who's reading your program listing what's happening on the stack before and after a command. The format is:

```
( before -- after )
```

The arrangement of values on the stack is shown both before and after the operation (note the space between the opening parenthesis and the start of the description). The actual operation is implied by the double-hyphen. Therefore, in an addition operation—just the + operation, not the extra stuff to display it and move the Mops prompt—you have two numbers on the stack before the operation, and you end up with a single number, the sum of those numbers, on the stack after the operation. That is, you start with n1 and n2 on the stack and end with the sum on the stack. The stack notation looks like this:

```
( n1 n2 -- sum )
```

This, therefore, is the description for the addition operation.

For the dot command, the description is:

```
( n -- )
```

because this command takes the topmost value from the stack and displays it on the screen. The value is removed from the stack in the process, leaving no trace of it after the operation.

In the CR command, there is nothing happening to values in the stack. It simply moves the prompt to the left margin of the next line. Because no stack operations are involved, the CR commands notation, then, is:

```
( -- )
```

The definition of every Mops word you define in a program should be accompanied by its stack notation. (Our convention is that we may omit the stack notation if it is (--), but only in this situation.) Thumb through the Glossary in Part IV of this manual to see how we have noted the stack actions of all the words in the Mops dictionary. While the notation will at first help you learn how Mops words work, it will also help you later when you start writing programs in an editor. The words and numbers in parentheses (with at least one space after the opening parenthesis) are not compiled into the program, so they won't add one byte to the size of your final program. The notations are there to aid you in tracing your program if you run into a snafu during program development. All in all, the stack notation is a handy shortcut to documenting your programs.

Note: Since anything in parentheses (i.e., starting with an open parenthesis followed by one or more spaces) is ignored by Mops, you don't have to type stack notation for words you define at the Mops prompt. Stack notation is strictly an aid for reading source code. In this tutorial, we often show the stack notation for words you define. The notation is presented to help you better understand the definition and show you how your definitions should look once you begin writing your own programs in an editor.

Here are Mops stack descriptions of the four basic arithmetic operations:

+	(n1 n2 -- sum)	Adds n1+n2 and leaves the sum on the stack.
-	(n1 n2 -- diff)	Subtracts n1-n2 and leaves the difference on the stack.

*	(n1 n2 -- prod)	Multiplies n1*n2 and leaves the product on the stack.
/	(n1 n2 -- quot)	Divides n1/n2 and leaves the quotient on the stack.

To newcomers, the stack order—the way in which numbers come out in the reverse order—may be confusing when it comes to subtraction and division, because in those operations, the order of the numbers is critical. If you want to subtract 4 from 10, you want to make sure that those numbers come out of the stack in the correct order for the subtraction operation to work on them. Fortunately, Mops saves you from performing all kinds of mental gymnastics in the process.

In the kind of arithmetic notation you learned in school, you write the problem like this:

```
10 - 4
```

and get the desired answer, 6. In Mops arithmetic, the order of the numbers going on the stack is the same. All you do is move the operation sign to the right. The problem becomes:

```
10 4 -
```

The same goes for division. The formula for dividing 200 by 25 changes from

200 / 25 to **200 25 /**

The four basic arithmetic operations are usable only on integers, that is, whole numbers like -2, 0, 3, -453, and 1002. Numbers with digits to the right of the decimal don't count. Don't worry, however, because Mops has plenty of ways to handle all kinds of numbers, as you'll see later on.

Experiment using the four simple arithmetic operations. Place one, two, three, and four integers (or more if you like) in the stack to understand how the operations make use of the numbers in the stack. Try them out now, and pay special attention to answers to division problems.

Everything should have worked well, except when you divided numbers that were not even multiples of each other. For example, if you divide 10 by 3, the Mops answer is 3.

```
10 3 / . CR
3
```

When you use the divide operation (/) in Mops, the remainder is lost forever. But Mops has two other operations that take care of the remainder for you.

/MOD	(n1 n2 -- rem quot)	Divides n1 by n2 and then places the quotient and remainder on the stack.
MOD	(n1 n2 -- rem)	Divides n1 by n2 and then places only the remainder on the stack.

Try out the 10-divided-by-3 example again, but this time using the /MOD operation instead of straight division (Remember! Mops does not distinguish between upper and lower case).

```
10 3 /mod . . cr
3 1
```

Notice now that both the quotient (3) and remainder (1) were returned to the stack (and subsequently printed out by two dot commands). Notice also the order of the two answers as they came out of the stack and how the order compares with the order of the /MOD stack notation above. The rightmost value in the stack definition, the quotient, was on the top of the stack and was therefore the first one to be printed out on the display.

Division involving negative numbers can be done in two different ways. In Mops we use the convention used by the Macintosh hardware, namely "towards zero" division. If the exact quotient isn't an integer, the quotient that the division operation gives will be the next integer towards zero. For example, -10 divided by 7 will give a quotient of -1, with a remainder of -3. The remainder will always have the same sign as the dividend (the first operand), unless it is zero.

Mastering postfix notation

If you're not particularly well versed in this reverse notation, called postfix notation, then it is important to recognize that complex math formulas need to be analyzed before they can be entered into Mops's postfix, integer arithmetic environment. For example, you may find yourself confronted with having to include the following formula in a Mops program:

$$\frac{1.25 * 12 * 50}{10}$$

If so, then Mops's integer arithmetic might seem like a stumbling block, and its postfix notation may seem worthless. But call upon simple algebra to convert everything to integers, and break up the complex formula into the same steps you would use to solve it with a pencil and paper. The Mops equivalent of this formula is:

```
5 12 50 * * 40 /
```

It's worth following what happens to the stack during a complex formula like this. First of all, to make the 1.25 an integer, multiply it and the denominator by four. Then put all three numbers to be multiplied into the stack. The first multiplication operation multiplies the topmost two numbers (50 times 12) leaving the result (600) on the stack. That leaves 600 on the top of the stack, and 5 below it. The second multiplication operation multiplies the two numbers remaining on the stack (600 times 5) and leaves the result (3000) on the stack. This result is the dividend (numerator) of the division about to take place. Now it's time to put the divisor (40) on top of the stack. Then the final operation, the division, divides the two numbers in the stack.

Don't be discouraged by all this concern over the stack. You'll learn in a later lesson that Mops provides you with two powerful tools —named input parameters and local variables—that let you substitute readily identifiable names for the values on the stack and use them at will. The stack will become almost invisible to you. It is important, however, to understand the stack fundamentals just the same.

End of lesson 3

Lesson 4

Mops and OOP

Armed with a basic knowledge of Mops's stack, you're now ready for an introduction to the language's real power: its OOP (object-oriented programming) nature.

OOP is such a popular topic these days, that a long introduction is probably unnecessary, so this introduction will be slightly briefer than in earlier versions of this manual.

The primary terms to concern yourself with at this point are:

```
CLASS
METHOD
OBJECT
MESSAGE
SELECTOR
```

To help you visualize the "big picture" of an object oriented system and what the relationships are among all the parts, we'll use an analogy.

Methods and Inheritance

Let's say you want to hire an accountant to prepare your income tax return. As a class of professionals, all accountants have a certain basic knowledge about accounting and manipulating figures. It is their fundamental job to adhere to generally accepted accounting principles when working on the financial records of a client. The methods they all use include calculating figures, cross-footing entries, checking calculations a second time, placing parentheses around negative numbers in a ledger, and so on.

But within that universe of all accountants, there are specialists. Some devote themselves to corporate tax work, others to accounting for self-employed professionals, such as doctors. No matter what the specialty, each shares the same fundamental knowledge of accounting as their colleagues in other specialties. That is, by virtue of being related to the class of accountants in general, they inherit many of the characteristics of all accountants. Most of their methods may even be the same, such as double-checking figures, using parentheses, and the like.

But some of their methods may be different. For example, one kind of accountant may specialize in handling financial records for corporations whose annual sales are in excess of $5 million. Another subgroup may do all kinds of accounting work, but its methods involve calculating the final tax form on a computer instead of calculating and writing entries by hand. In the case of each of these subgroups, their predominant methods are the same, but with minor variations in certain methods. Therefore, while each subgroup—subclass—of specialty accountants is a class unto itself, each retains many ties to the larger class of all accountants.

A yet smaller segment of a subclass of accountants, however, can have its own special methods. For example, there could be a small subclass (actually a subclass of a subclass) of computerized accountants who bring a portable computer along and perform the work only at the client's place of business. But even this sub-subclass can trace its methods back through all levels of the class hierarchy, which might look like the one in Figure I-2.

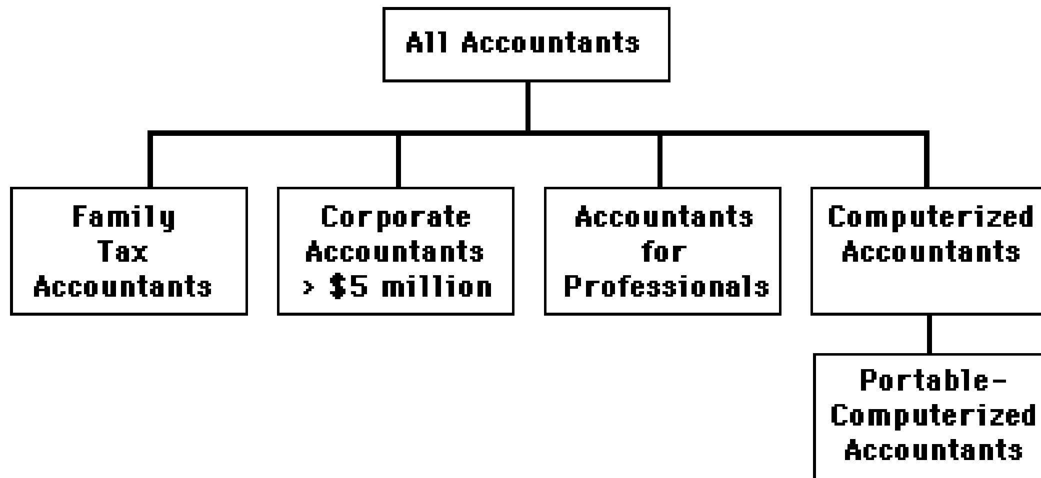


Figure I-2

Objects and Messages

So far, we've been talking only about classes of accountants, not the actual people who do the work. The accountant you select to do your taxes, say his name is John, would fall into one of the subclasses that best meets your particular tax needs. For the sake of this example, let's say that John is a member of the class of accountants that works with family tax planning and tax return preparation. In other words, John is an "instance", or an actual, physical example—an object—of the class of family tax accountants. When you summon John to do your taxes, he automatically brings with him the ability to perform all the accounting and tax preparation methods that belong to the specialty subclass he belongs to, as well as all the methods he inherits by belonging to a hierarchy of accountant classes. He may not have to summon absolutely every method for your tax job, but they're in his background just the same (see Figure I-3).

To get John going on your tax return, you give him the appropriate instructions, including all the figures he needs and the final go ahead. In other words, you give him the message, "prepare the tax return based on my figures."

When John receives this message, he knows that the figures you provide are the parameters to be passed to the methods he will be using to calculate your taxes. He also knows, according to the methods in his background, that "prepare the tax return" means he should do certain things, like organize the figures, obtain copies of each tax form necessary, and so on. The "prepare the tax return" part of the message is a selector in that it tells John what method—of the many methods in his background—to proceed with first. Even within that very first method he performs, some of the individual steps, such as organizing the figures, may be inherited from the superclass of all accountants. One or more of those steps, however, may be unique to his subclass of family tax preparers.

Now, let's say that at the office you are responsible for hiring an accountant to do the company tax return. Because John is a specialist in family tax planning, you wouldn't want to select him. Instead, you hire Marvin, because he comes from a class of corporate tax accountants.

To Marvin, you give almost the same message: "prepare the tax return based on the corporate figures." Marvin receives the same message as John, but because the methods in Marvin's class are not identical to John's, a different process takes place in the preparation of the return. Some of Marvin's steps may be the same as John's, because they share the same steps with all accountants, but others will be unique to Marvin's subclass. And the corporate figures you give Marvin, even though many will have the same names as the personal figures (income, medical expenses, tax credits, interest deductions), they will in no way be mixed between returns. Only your family's figures will be in John's return; only the corporate figures will be in Marvin's. Despite John's and Marvin's common heritage of accounting methods, they work completely independently of each other.

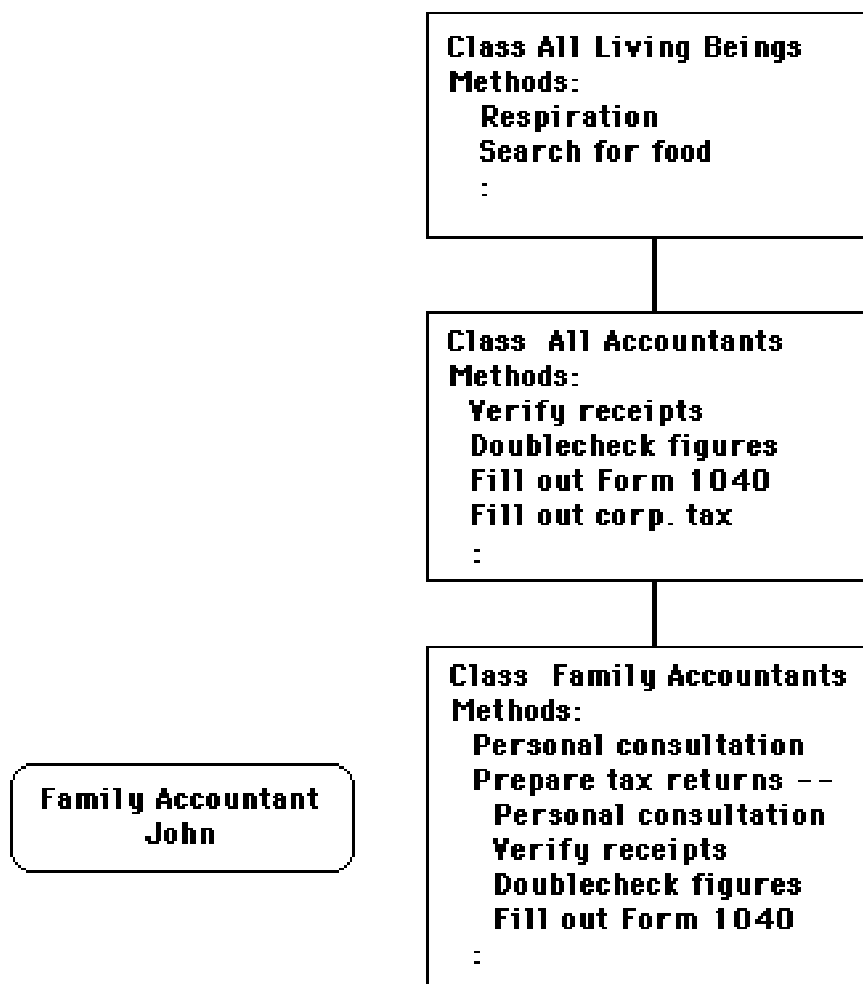


Figure I-3

The same would be true if you hired a colleague of John's class to prepare your mother's tax return. If his name is Percival, you can give Percival the same message and your mother's figures, and there would be no interference among the three returns you have in the works.

If this accountant example were a true object oriented system, the class of all accountants would, itself, be based on another, all-encompassing superclass—something like "all living beings." In other words, there must be a primeval class from which all classes are derived, and all the primeval methods apply down the line, as long as they haven't been modified by a subclass. Therefore, even though John doesn't think about it, he breathes, his heart beats regularly, he seeks nutrition periodically, and so on. If you send the message, "John, hold your breath for 15 seconds," the method for breathing would not be found in either of the accountant classes to which John belongs, but rather in the primeval class of living beings. It's possible, nonetheless, for John to reach back through the

hierarchy of classes to that primeval class and make a change to the method that controls his breathing.

Classes and subclasses are defined by the methods that dictate how an object is to behave. A subclass inherits all the methods of its superclass, and adds to or modifies the superclass' list of methods, if necessary. An object is a singular instance of a class or subclass. An object is capable of performing all operations specified by methods in its class and its superclass.

For an object to do any work, it requires that a message be sent to it. The message must contain a selector, which the object matches with one of its possible methods. Any data (parameters) passed to the object inside the message remain the private property of that object.

In Figure I-4, when we send the message, "John, prepare tax return with my figures," John matches the selector "prepare tax return" with the methods in Class Family Accountants. This method is, in turn, defined by a method from its own class (e.g., Personal Consultation) and by methods that the subclass inherits from its superclass (e.g., Verify Receipts, Doublecheck Figures, and Fill Out Form 1040), as shown in Figure I-4.

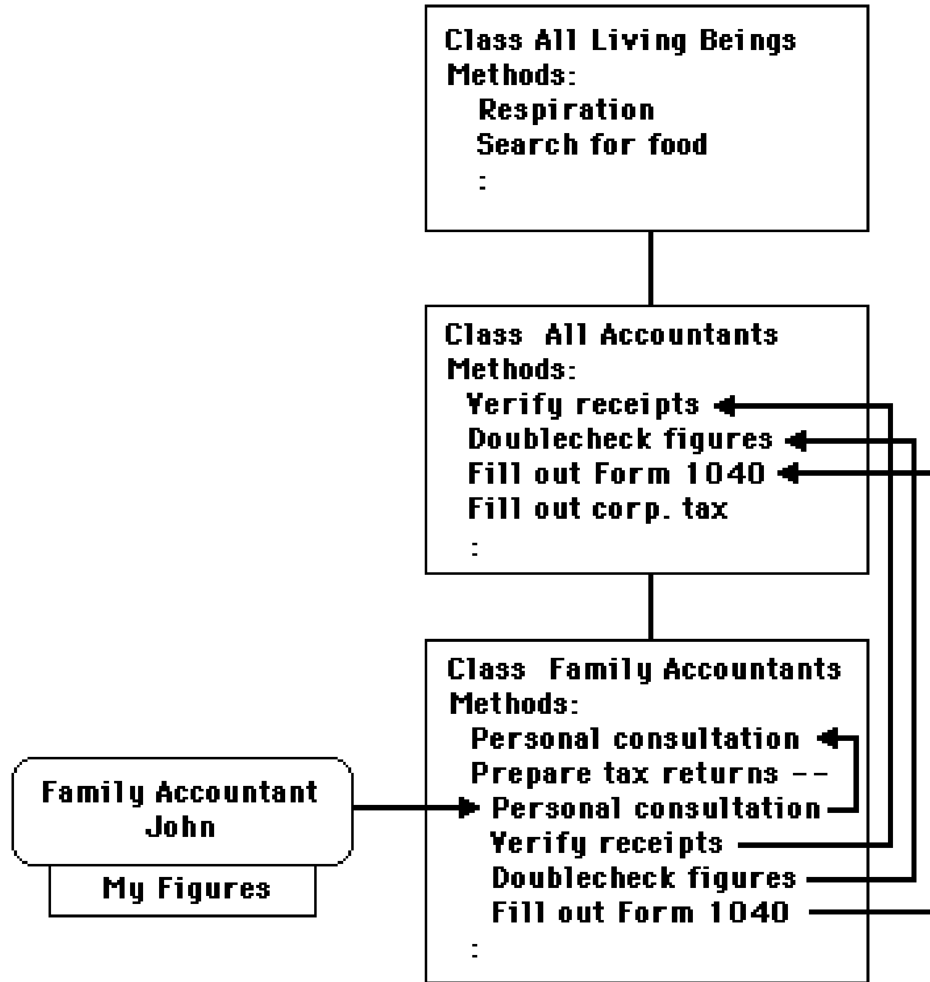


Figure I-4

When you send the same selector to Percival, but with your mother's figures, Percival follows the same procedures as John, but never sees your figures, which John has to himself (see Figure I-5).

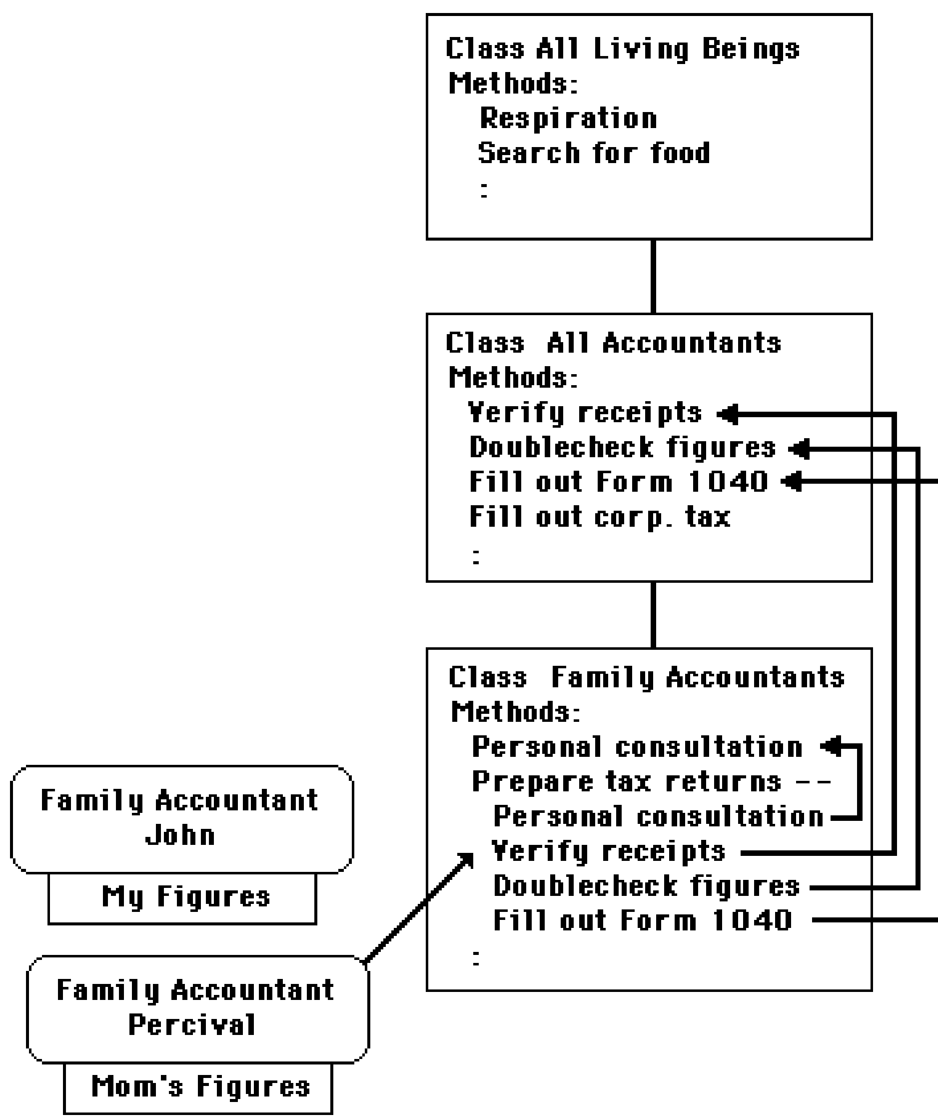


Figure I-5

End of lesson 4

Lesson 5

Mapping class-object relationships

An object oriented language like Mops builds programs around the same kinds of relationships as portrayed in the accountant metaphor. Class definitions play a central role in the structure of a program. As such, the most important early step in planning a Mops program is to visualize what the main objects—the actors—in your program will be doing. Because the Macintosh is capable of recreating on-screen metaphors for so many different real-world objects—a bank book, an artist's canvas, a calendar—it is best to devise classes of Mops objects that bear a behavioral resemblance to the real-world items. Once you've determined what the program's classes will be, it's time to start writing the program by defining those classes with methods. Then create objects of those classes. Finally, write the messages to those objects that set the program in motion.

Let's take the first steps in applying class-object relationships to a Mops program by defining a class that is capable of drawing rectangle objects on the screen. At the same time, you'll also be introduced to the way Mops programs really look. Pay particular attention to the physical structure of program listings—indentions, spacings, capitalizations, and the like. While Mops is pretty loose about how you make your programs look, the ways prescribed hereafter will help make your code more readable. Also consult chapters 3, 5, and 6 in Part II, for in-depth discussions of this and related topics.

Defining a class

As you may have noticed in the accountant class metaphor, each class was defined by what amounts to a series of behavioral rules or procedures that are to be followed whenever an object of that class is called into action. Defining a class, then, entails establishing those rules and procedures: the methods.

Most classes also have information—data—associated with an object of the class. For example, the class of Family Accountants can dictate that every accountant of its class should be paid for his work. Every family accountant (John and Percival, for instance) carries with him a figure for his hourly rate. The class definition merely states, "Thou shalt have an hourly rate." When the objects are created, the rate is plugged into that variable. Importantly, John and Percival can have entirely different hourly rates, because they hoard their own data to the exclusion of other objects in the same class. One of their methods would retrieve the rate, multiply it by the number of hours spent on your taxes, and send you the bill.

Let's see what it's like to build a Mops class called Rect, which will define all the procedures for creating rectangle objects.

In the Macintosh environment, a rectangle is defined by two points on the screen: the locations of the top left and bottom right corners of the rectangle. In other words, for every instance of a rectangle on the screen, an object of class Rect will need numbers to fill in these two variables. These variables, then, are called <u>instance variables</u> (ivars, for short). They are the holding places in an object's definition for the requisite data—the two points—required before a rectangle can be drawn.

The class definition up to this point looks like this:

```
:class  RECT  super{ object }
        point     TopLeft
        point     BotRight
```

Notice several things. First of all, the beginning of a class definition is :class (pronounced "colon class"), with no space between the colon and the word "class". There is at least one space or a tab between :class and the name of the class. We have put RECT in capitals to make it stand out, since this is where it is defined. However, this is not necessary, since upper and lower case are treated the same by Mops. You can use whatever style of formatting you prefer.

On the same line as the name of the class is a reference to the superclass from which the class Rectangle is derived. This reference takes the form of the word super{ (no space between super and {), then the name of the superclass, then a }. These three items are separated by spaces or tabs, as for all Mops words. (We will see later that it is possible for a class to have more than one superclass—this is called multiple inheritance. We won't go into the details of this now, except to say that if there is more than one superclass, these are put one after the other before the }, again separated by spaces or tabs.)

Although in this example the superclass name is "Object", this should not be confused with the general use of the word object, as it is applied in the Mops system, where it refers to all objects generally. In this one special case, "Object" is a class. This may seem a little confusing, but it is actually because we do use the word "object" in a general way, that we have named this special class "Object". This is because all classes in Mops can trace their inheritance to class Object. Thus, class Object defines the behavior appropriate to all Mops objects. This is why the name "Object" is appropriate for this class.

By its inheritance, then, class Rect has at its disposal all the methods defined in class Object. If you are interested, you could check the source code listing for class Object (located on the Mops disk as the source file labeled Object) to see what methods are defined in class Object.

The instance variables tell Mops to reserve memory space in the data area of any object created from this class. The amount of space to be reserved is determined by the characteristics of the instance variables—which are, themselves, defined by other classes. Here, the instance variables (ivars) are named TopLeft and BotRight, both belonging to the class Point. It would not be possible to create ivars TopLeft and BotRight in class Rect if class Point had not been previously defined—however, class Point is one of Mops's many predefined classes.

(For procedural language buffs, a key to understanding the object orientation of Mops is that as you follow the threads through the dictionary in the next few steps, you are not watching straight execution steps. Rather, you are building a framework that will reside in memory as a kind of potential energy that is released only when a message is sent sometime later in the program.)

To understand what the rules and procedures are for the Point objects (TopLeft and BotRight) created inside class Rectangle, you can look up the Mops source code for the class Point (located in the QD source file in the "Toolbox classes" folder). The class definition looks like this:

```
:class  POINT     super{ object }
record
{     int  Y            \ Vertical coordinate
      int  X            \ Horizontal coordinate
}

:m GET:           get: X  get: Y   ;m
:m PUT:           put: Y  put: X   ;m

;class
```

We'll explain all of this shortly, but the main thing to notice first of all is that this class, itself, uses two more ivars, X and Y of class Int (integer). They specify the data area inside any object of class Point. In other words, any object created from class Point will need two integers to fill the cells reserved for data. Class Point was designed in this way so that two values, representing a coordinate point, would be conveniently coupled together whenever a Point object was created.

Notice, too, that we've started adding plain English comments about the code as a way of documenting the program. There are three ways of specifying comments in Mops:

```
( this kind comment continues to a )

\ This is another comment, which extends to the end of the line

(* This kind of comment
can go over several line,
   (* and can be nested *)
*)
```

Note that (, \, (* and *) are Mops words, and so must be followed by a white space character. Thus if you had

```
(this isn't a comment)
```

Mops would try to recognize a word "(this", and wouldn't treat this line as a comment.

We'll come back to the rest of the statements in this class Point in a moment. First, we must search once more, but this time for the class definition of class Int, because the data of class Point consists of ivars Y and X that have the characteristics of class Int. Class Int is defined in the file Struct in the Mops ƒ folder.

```
:class      INT   super{ object }

      2 bytes data

:m PUT:     inline{ obj w!}  ^base w!  ;m

;class
```

class Int is another one of Mops's predefined classes. It states, first of all, that its superclass, like many in Mops, is class Object. Next, it states that two bytes (16 bits) of data are set aside for each value whenever an integer object is created. The third line is a method of this class (preceded by :m and ended by ;m). The message inside this method definition stores an integer in a special area of memory (don't worry now about details of this method definition for now).

Going back to the class Point definition, the method in its fourth line is a single instruction for Mops to store both the X and Y coordinates in memory. Therefore, every time one of the ivars (TopLeft or BotRight) is given two numbers for an X,Y coordinate, the entire coordinate is stored by one PUT: message.

Returning at last to class Rect, then, the list of two instance variables for this class means that an object of class Rect holds reserved space for all the data needed by the two instance variables. And, as you've seen, the two instance variables will require a total of four integers to signify the opposite corners of the rectangle's boundary.

Next in the class Rect definition come two methods:

```
:class  RECT  super{ object }

         point     TopLeft
         point     BotRight

:m PUT:  ( l t r b -- )   put: BotRight  put: TopLeft  ;m
:m DRAW: ( l t r b -- )   addr: self   call FrameRect  ;m

;class
```

As detailed in the stack notation, the first method, put:, requires four integers on the stack (here signified by the letters l, t, r, and b) before an object executes it. The first two integers (the ones on the top of the stack) are put into the object's BotRight reserved cells as soon as the put: BotRight message finds the definition of the put: method in BotRight's class, class Point. The second two integers are placed in the object's TopLeft cells as the result of the put: TopLeft message in this put: method. In other words, when an object of class Rect receives a message consisting of the put: selector, the object searches its own class for the corresponding methods definition. The method sends messages of its own to objects of other classes, and so on back through a chain of classes and objects until a method is reached that is defined purely in Mops words (as in the put: method in class Int). All the actions taken by this series of messages affect only the private data of the Rect object that received the message.

The second method, draw:, calls a Macintosh Toolbox routine, named FrameRect, to draw the rectangle according to coordinates currently in the data cells of the object being drawn. The data, of course, must be in the proper order that FrameRect expects. FrameRect and most other Toolbox calls seek the address of an object's data. This is obtained by the addr: self message in the draw: method. This address is then passed to the Toolbox call.

What we have so far, however, won't work properly if we try using the draw: method. This is because by declaring topLeft and botRight as we did, we have made them proper Mops objects. The problem with this is that Mops objects have 8 bytes of extra information at the start, which Mops uses to keep track of certain things including the class of the object. However the Toolbox doesn't know anything about Mops objects, and just expects 2 bytes each for topLeft and botRight, with no extra bytes present. Accordingly we have to have a way of omitting this extra information, and we do this with the record { ... } syntax, as follows:

```
:class  RECT  super{ object }

record
{     point TopLeft
      point BotRight
}

:m PUT:  ( l t r b -- )   put: BotRight  put: TopLeft  ;m
:m DRAW: ( l t r b -- )   addr: self   call FrameRect  ;m

;class
```

Any ivars declared as part of a record won't carry any extra information. This will limit some of the things you can do with these particular ivars, as you might expect, since Mops doesn't have the extra information available. But as we'll see, this isn't a very serious restriction.

To end the class definition, use ;class (pronounced "semicolon class").

Finally, you can format your class definitions (and all your code for that matter) however you like, as long as at least one space or tab separates Mops words. The formatting we use here in the Manual and in the Mops source code is quite readable, so we recommend something like it. Plenty of white space and comments are always a good idea, as it will greatly help anybody elsewho has to read your code and understand it (and you yourself for that matter, in a few weeks' time when it's no longer fresh in your mind). But in the end the choice is up to you.

End of lesson 5

Lesson 6

Objects and their messages

Now we come to creating an object of class Rect and sending messages to that object so it can select the methods to execute. To create an object of class Rect, the syntax is simply the name of the class followed by the name you want to assign to the object. For an object named "box1" of class Rect, the statement would be:

```
Rect  BOX1
```

That's all there is to it. By creating this object, you have added a new Mops word, "box1," to the dictionary in memory. You can visualize the object in memory to look like Figure I-6:

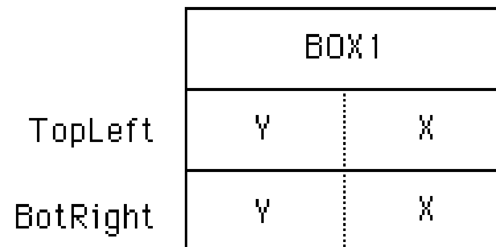


Figure I-6

Zeros are placed in the instance variable cells when the object is created, and they are holding space for numbers whenever the object receives a message to put data there.

When you type a Mops message in a program, it has three parts to it: the parameters, selector, and receiver.

Parameters are the numbers to be passed to the operation. They are placed on the parameter stack just like parameters in Lesson 1. Not all messages have parameters, of course. Some operations don't require any numbers be passed to them.

The second part, the selector, is actually the name of the method containing the operation you want the object to perform. In other words, the object "selects" which method of its class is to be put to work; the object matches the message's selector with the method in the object's class (or up the superclass hierarchy if there is no match in the immediate class).

The last part of a message, the receiver, must be the name of an object. It is the "thing" on which you want to perform the operation specified by the selector. In the accountant metaphor, the receiver is the name of the accountant who is to "prepare the returns."

Since box1 is an object of class Rectangle, you can send a message to it that selects one of the methods defined in class Rectangle. If you send the message:

```
300 20 400 100 put: box1
```

you put the coordinates 300, 20 and 400,100 into the data cells reserved for TopLeft and BotRight in the box1 object. After all, that's what the put: method in box1's class does: it places two sets of two parameters into an object's data cells.

If, at some future time, you create a new object of class Rect, called "box5," box5's data cells would be empty at first. A separate put: message would have to be sent to box5 to place box5's coordinates in that object's data cells. This is how objects maintain private data.

To draw the objects on the screen, you need to send another message, one that calls upon the draw: method of class Rect. The message would be:

```
draw: box1
```

If you were defining class Rect from scratch, you could also define a new method that combines the functions of two methods into one. Then, a single message would take care of both the put: and draw: methods. For this to happen, you need a way for the new method to look up the methods in the same class. That's where a message receiver called "self" comes in handy. With the new method (place:) the class looks like this:

```
:class  RECT  super{ object }

record
{     point TopLeft
      point BotRight
}

:m PUT:  ( l t r b -- )   put: BotRight  put: TopLeft  ;m
:m DRAW: ( l t r b -- )   addr: self   call FrameRect  ;m

:m PLACE: ( l t r b -- )    ( draw at new coordinates )
      put: self  draw: self  ;m

;class
```

The place: method contains the messages, "put: self" and "draw: self." The put: self message is saying, "Do to the current object everything that the put: method in this class does." The same goes for draw: self. If you had intended one of these messages to look up a method in Rect's superclass, the receiver would have been "super," as in put: super.

Something important happens when you have the put: self message inside the place: method. The place: method now expects to find four integers passed along with any message bearing its selector, just like the actual put: method that executes the storage command requires four integers. Therefore, to both locate and draw Box1 on the screen, you would send the message:

```
12 10 100 50  place: box1
```

If you want to try this, you'll have to have a window to display box1 in, as you did in the Intro. So first copy the above Rect definition to the Mops window (either by typing it in or by copying and pasting it). Then select the whole of the definition (by dragging with the mouse). Then hit the Enter key. This will cause all the selected text to be executed. In this case, since the code is a class definition, the result of executing the code will simply be to define the class Rect. Nothing will seem to happen, but the definition for Rect will have been entered into Mops' dictionary.

Now type and execute this:

```
window ww
test: ww
```

Click back on the Mops window and move things around so you can see both the Mops window and ww, then type and execute

```
rect box1
set: ww  12 10 100 50  place: box1
```

and your Rect should appear in the window ww.

Another way of doing this would be to get the above code into an editor, save the source file, and load the file into the Mops.dic window. If you want to do this now, open your editor. In a clean edit window, type the class definition and the other lines of code above, or copy and paste it from here. Each line should be terminated by <RETURN>—don't let your editor wrap lines around automatically. When finished, select Save As... from the FIle menu, and assign a short, recognizable name to the file, like "rr." If your editor uses different document formats, you should use TEXT.

Close the editor window to return to Mops.dic. Load the file into Mops.dic by selecting Load from the File menu and choosing your file.)

The advantage of using an editor rather than the Mops window directly, is that you now have a file available for later use. But if you are just experimenting, it is quicker to use the Mops window, and you can still select the text and copy and paste it into a file in an editor's window at any time.

Summary

Before taking one more step, let's summarize. Creating a Mops program entails the following steps: defining classes; creating objects that are instances of those classes; and then sending messages to those objects. Building a hierarchy of classes starts with the broadest class and works toward the more specific, with subclasses inheriting the characteristics of their superclasses.

To help you visualize the structure of the program example detailed in this chapter, look at Figure I-7. It graphically portrays the relationships between the classes and objects discussed above.

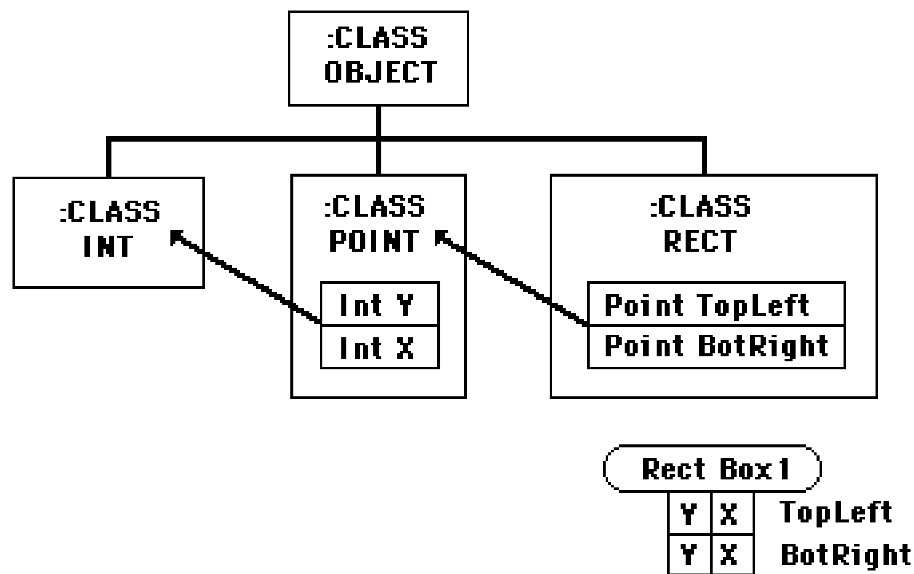


Figure I-7

Given this framework, when you issue the message 300 20 400 100 place: box1, the parameters fill box1's data cells held in reserve when box1 was created. The characteristics of the data had already been determined by the ivars TopLeft and BotRight; the characteristics of those ivars had been likewise determined by the ivars X and Y, which, in turn, had been defined by the methods of their defining class, class Int.

Therefore, you probably recognize that the relationships in Mops classes and objects are on multiple levels. On the one hand, you have the relationships between superclasses and subclasses. On the other hand, you have the relationships between ivars and their defining classes. Both relationships cascade through the hierarchy of a Mops program independently of each other. That will become even clearer as we make one further extension to the example above.

End of lesson 6

Lesson 7

Modifying a Mops program

We're going to add another class. This one, however, will be a subclass of Rect because our goal is to produce an object that draws a rounded rectangle. A rounded rectangle requires the same parameters as a rectangle with the addition of one more, the size of the ovals whose curvature rounds the corners. The oval's dimensions are determined by the number of pixels high and wide as in Figure I-8.

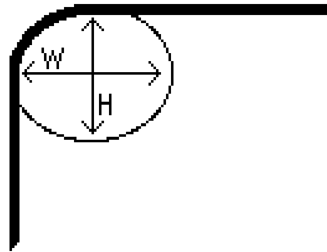


Figure I-8

The Toolbox call, FrameRoundRect, expects these dimensions as a 4-byte data cell—a construction that Mops handles well as a Point instance variable.

Since a rounded rectangle has so much in common with objects created by class Rect, the logical addition would be a subclass of class Rect called, class RoundRect. It needs one additional piece of data, which we've named ovalsize. The data will be converted from height and width figures to a point, which the Toolbox expects. Therefore, the instance variable for class RoundRect will be Ovalsize of the class Point. By virtue of its inheritance from class Rect, then, an object of class RoundRect will have a total of three ivars: TopLeft, BotRight, and Ovalsize. TopLeft and BotRight refer to the corners the RoundRect would have if it wasn't rounded—that is, the intersection points of the lines on which the sides lie. These points, of course, will actually lie outside the rounded corners.

Next, the class needs a method to store the values its object receives from messages. The ovalsize value for this class will be stored by way of an init: method inside class RoundRect. The values for the coordinate points (TopLeft and BotRight) can be initialized just like the points in class Rect, because the put: method from class Rect is still available to an object of class RoundRect. Simply define the new part for RoundRect that stores the ovalsize, and pass the burden of coordinate storage back onto the put: method in the superclass.

class RoundRect needs a draw: method to act on the values stored in an object created from its own class. In this particular draw: method, ^base retrieves the "base" address of the current object, i.e. the address of the beginning of the object. In this case it is the address of the rectangle coordinates. The Toolbox uses this address to locate the values it uses as parameters. Next, the ovalsize values are put on the stack in a form the Toolbox expects (using the int: method of class Point), and then the proper Macintosh Toolbox routine (FrameRoundRect) is called to do the actual drawing on the screen.

The subclass definition looks like this:

```
:class  RNDRECT  super{ rect }

        point  OvalSize
```

```
 :m INIT:  ( w h -- )   put:  OvalSize   ;m
 :m DRAW:  ( -- )       ^base  int: ovalSize  call frameRoundRect  ;m

;class
```

That's all that was needed to add an entirely new kind of object to Mops.

Once the class is defined, it is now ready for the creation of an object like:

```
RndRect CYNTHIA
```

To draw this object in a window ww, as we did in the previous lesson, you can do this (using either the Mops window or an editor):

```
20 30 init: Cynthia
20 20 100 60 put: Cynthia
window ww
test: ww
draw: Cynthia
```

The 20 and 30 values are the width and height of the oval in the rounded corners. The PUT: method is inherited from the superclass Rect, and sets the rectangle coordinate. Look how the addition of this subclass works within the structure of the overall program in Figure I-9.

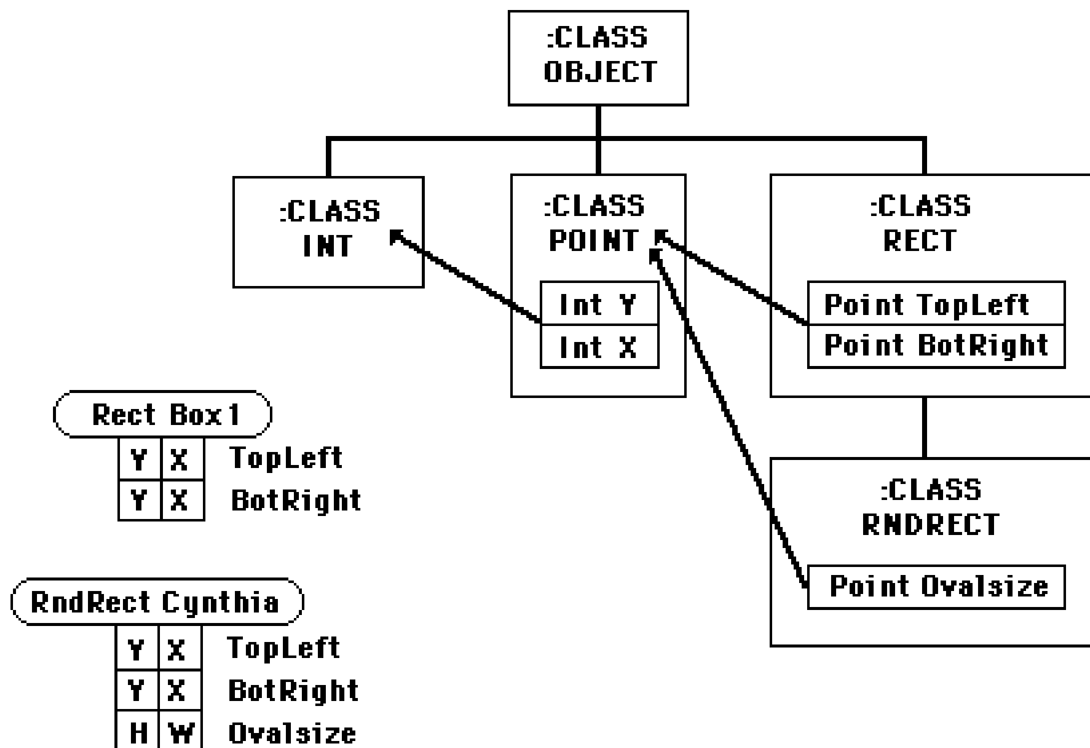


Figure I-9

Next, you'll be introduced to the powerful building blocks of Mops: the predefined classes.

End of lesson 7

Lesson 8

Predefined classes—an introduction

Mops comes with a number of predefined classes that provide you with a strong foundation upon which to build your programs. The more you know about these classes—especially their methods and the powers of the objects they create—the more comfortable you will be in designing your programs. Mops in many ways is like an Erector Set—we provide the pieces, you provide the imagination to turn those pieces into a usable program.

Predefined classes serve an important function in Mops. They insulate you from the concerns of extensive stack manipulations and other memory maintenance chores for frequently used Mac Toolbox operations: windows, menus, graphics, disk file manipulation, and dozens more. In fact, most of the complex stack stuff is handled within the predefined Mops kernel, so even the methods in the predefined classes will be largely understandable to you by the time you're finished with this tutorial.

What this all means is that while you send comparatively simple messages to objects derived from those classes, you are automatically performing very sophisticated memory manipulations not far different from those that an assembly language programmer would use. You are also left with fewer concerns about making your program Mac-like, since the predefined classes point you in the right direction from the very start.

You will soon want to begin scanning through the source code of the predefined classes. While much of the code is already compiled in the disk file Mops.dic, all the Mops source code is supplied, in the folder "Mops source". You can view and print the source code files using any text editor or word processor. Eventually, you will find it helpful to keep a printout of all the source code in a looseleaf binder. At your earliest convenience print out the text files in the System, Toolbox, and Demo classes folders within the Mops source folder. Put the printouts in alphabetical order according to the name of the files. You will then have a much easier time tracing the hierarchy of a class chain or finding the details about a particular method of a class.

As you can see within the Mops source folder, Mops predefined classes are divided into three groups. One group, called Mops System classes, consists of classes that are not necessarily specific to the Macintosh. The System classes control things like file manipulation, basic data structures (integers, variables, arrays), and other computer housekeeping tasks. These classes, of course, have been designed to work specifically with the Macintosh, but they work largely behind the scenes, since they don't directly affect the way you and the computer communicate with each other.

A second group, called Toolbox classes, are those that make the connection between the programmer/user and the graphic elements of the Macintosh. "Graphic elements" is a broad category that includes such things as menus, windows, text input, mouse manipulation, and program control via the mouse or keyboard. The Toolbox classes are the highly visible, "show biz" classes of Mops.

The third group, Demo classes, consists of demonstrations files.

Most of the predefined classes in both categories are subclasses of a kind of Master Superclass, called class Object. While class Object, itself, is a subclass of yet another superclass, class Meta, you won't have to concern yourself with that particular relationship. Just think of class Object as the ultimate superclass of all classes, and you won't go wrong. Class Object is predefined in Mops, in the Mops source file "Class".

Data structure classes

Among the most used predefined classes are several that are grouped into a cluster called "data structures." Figure I-10 shows the organization of the Mops data structure classes, which are listed in the files called "Struct" and "Struct1."

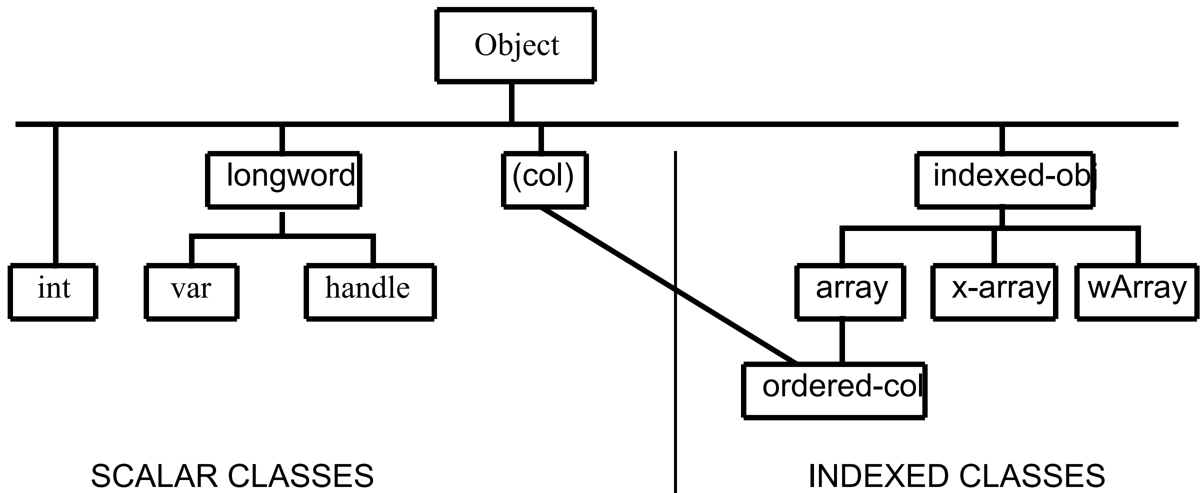


Figure I-10

These classes form the basis of much number and string (text character) storage and manipulation inside a Mops program. In the rectangle example in Lesson 6, you already saw how instance variables of one basic data structure class, Int, were used as components of coordinate point objects, which were, in turn, used as components for a rectangle object.

The classes to the left of the dotted line in Figure I-10 are called scalar classes because they reserve a fixed amount of memory space for each instance of their class (just like a ruler marks a fixed area according to its "scale"). An integer object, for example, always has two bytes reserved for data, whether or not both bytes are filled with data when an integer object is created.

To the right of the dotted line in Figure I-10 are a group of indexed classes. You can tell from the names of most of them that these classes provide the rules for setting up arrays in Mops programs. An indexed array is a convenience that helps your program reach into a list of data in memory and pick out desired pieces. If you consider that an array object might look something like Figure I-11 in memory:

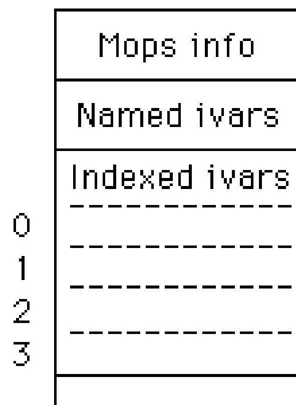


Figure I-11

you'll notice that some data cell have reference numbers attached to them. Each number is an index—like an index tab in a three-ring binder—to that data cell. It is much easier to reference an object's data by an index number than it is to cite the specific address in memory for the piece of data your program needs at a given moment.

The differences between the various indexed classes in Figure I-10 include the number of bytes each data cell is to contain (1, 2, or 4) and other considerations discussed later.

You'll also notice that the class Ordered-col has two lines leading to it—this is because it has two superclasses, that is, it uses multiple inheritance. The class (col) defines some methods that all "collection" type classes need, regardless of the size of their elements. Then the array superclass specifies that the class Ordered-col has 4-byte elements.

The class (col) is an example of a class which doesn't have any objects of its own. It is just a convenient way of defining a group of classes, where these classes have a number of methods in common. Rather than repeat the definitions of these methods in each class definition, we "split them off" into a single superclass in which they can be defined just once. We call this kind of superclass a generic superclass. The class Longword is another example of a generic superclass. It is useful for defining methods common to the classes Var and Handle, but it doesn't have any objects of its own.

Other predefined classes

Another group of classes that gets a workout is the one that links you to QuickDraw, which is Macintosh's powerful tool for accessing many of its graphics features. Figure I-12 shows the QuickDraw classes and the superclasses from which they were derived:

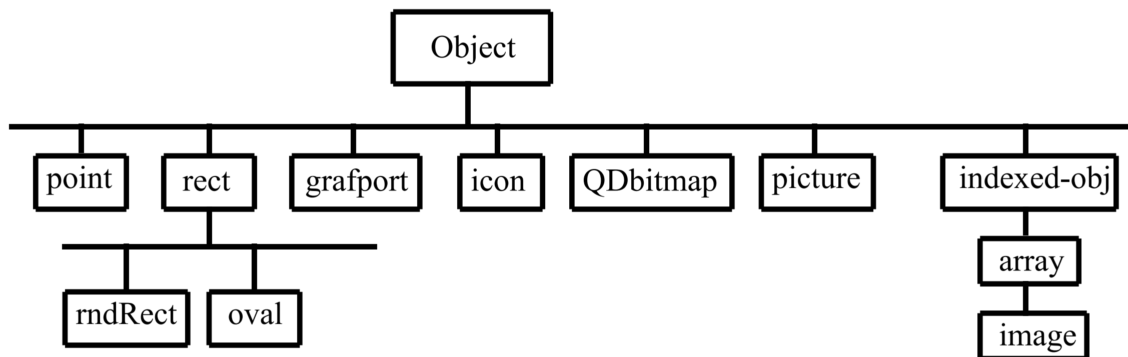


Figure I-12

Other graphics oriented classes include those that help you create windows, menu bars, and menus, plus a class called Control that reigns over reactions to clicking the Mac mouse on buttons and scroll bars. In addition, there are numerous predefined classes and objects that give you shortcuts to opening and closing files, sending output to the printer, producing sound, and other functions. Part III of this manual contains in-depth explanations of Mops's predefined classes. You will look to these reference sections often once you have completed this tutorial.

End of lesson 8

Lesson 9

Defining new Mops words

We said earlier that you can add words to the Mops dictionary while building a program. In fact, that is largely what programming in Mops is all about. Class names, method names, and object names become part of the dictionary in your program. Defining new words in Mops also lets you write your own shortcuts by defining one short and simple word to take the place of several commands that otherwise require more typing precision.

Special Note: Unless you save to disk the dictionary you've assembled for a program, the words and definitions will not be remembered by the Mac if you quit Mops. In the remainder of this tutorial, you will be defining new words that pertain only to this tutorial. If you wish to save the current state of the dictionary at the end of a lesson, then choose Save As... from the File menu and type in a name for the file, or else you may choose Save if you have already given it a name. You will be able to recall the dictionary at a later time by double-clicking its icon.

The first definition exercise will be to define a new word that takes care of the symbols in a simple addition problem. The new word is "add," although you could choose any word not already in the Mops dictionary.

The safest way to doublecheck that a new word you want to define is not in the dictionary, is to issue the "tick" command with the word you want to test for. In Mops, a tick is an apostrophe. By typing apostrophe, space, and the word you're testing, Mops searches the dictionary for the occurrence of that word. If the word is in the dictionary, tick will leave a number on the stack (the location in memory of the word's definition). But if the word is not in the dictionary, the message "not found" appears on the screen, and you're in the clear to define a word with that name. (Remember to put <ENTER> at the end of every line you type):

```
' window .
9557892
' twindow .

Error # -13 : undefined word
' twindow .
        ^
Current object:  TW    Class:
Stack:  Depth 1
9568706  $9201C2
Return stack:  Depth 43
9513434  $9129DA ?NOTFOUND
9513710  $912AEE '
9502126  $90FDAE (no name)
```

You define a new Mops word by typing a colon, a space, the name of the new word, one or more spaces or tabs, the sequence of values and/or commands to be performed when you use that new word, and then a final semicolon, indicating the end of your new definition. This kind of Mops definition is called, aptly enough, a colon definition. Notice especially that although class and method definitions don't want a space between the colon and either class or m, these standard colon definitions do. It might be easier to think of :class and :m as being special purpose colon definitions.

Here's an example that defines a new word, "add," which will perform the addition of two numbers on the stack, display the results, and move the Mops prompt to the left margin of the next line. (Remember, if you're typing the definition from the Mops prompt, you don't need to type the stack definition. We just include it here for completeness.):

```
: add  ( n1 n2 --  )  + . cr ;
```

The + operation expects to find two numbers on the stack. Therefore, to use your new word, you would type two numbers (which go onto the stack) and then the new word:

```
2 6 add
8
```

A good exercise at this point would be to define new words to simplify the other basic arithmetic operations.

The return stack

As we have seen, a Mops program basically consists of a sequence of words, and messages sent to objects (which cause a method to be executed). The definitions of these words and methods can contain many other words and messages. If you think about what must happen when Mops is executing one definition or method, you can see that when it has to go and execute other words or methods, it will then need to come back to where it was. It needs to mark its place in some way. The way this is handled is with a second stack, called the return stack. When Mops has to execute something somewhere else, it saves its current position as an address on the return stack. In that other place, if it has to go yet somewhere else, it pushes the new address on to the return stack as well. This is how words or methods can call other words or methods which can in turn do the same, down to a great depth. And by using a second stack, all these return addresses on the return stack don't interfere with items on the parameter stack.

Normally you won't need to worry about what's going on on the return stack. When there's an error, however, it's usually very useful to know what the program was executing, and where it had come from before that. Our error dump includes a dump of the return stack. Mops puts some other items besides return addresses on the return stack, but for items there which look like return addresses from within a word, the dump will include the name of the word. If you look at the dump just above, for example, you can see that the address of ' (tick) is underneath the address of ?NOTFOUND. This shows that ' called ?NOTFOUND and it was at that point that the error was detected. (?NOTFOUND in fact checks the result of a dictionary search to see if it was successful).

We can't display the names of methods in this way, however, since the names of methods aren't stored in a readable form. However just seeing the names of the words which were executing at the time of an error can give you very useful information.

Named input parameters

We're going to make Mops a little easier for you by reducing what may be undue concern about the way numbers are stored on, and recalled from, the parameter stack. Whenever you define a new Mops word, Mops lets you assign names to the parameters that are passed to it. After that, you needn't worry about the stack or the order of the numbers: when you need them for operations, simply call them by name.

As an example, use the arithmetic problem cited earlier. If you recall, the problem was:

$$\frac{5 * 12 * 50}{40}$$

To calculate this with Mops previously, you had to multiply the three numbers in the numerator, and then place the denominator on the stack before dividing. Watch how this is simplified in a definition that performs the math with named input parameters:

```
: formula  { denom n1 n2 n3 -- solution }
     n1 n2 n3 * * denom /  ;
```

The magic of named input parameters takes place inside the curly brackets. The syntax is deliberately similar to a stack comment, because it is in fact a kind of stack description. So, in this case, whenever the formula is executed like this:

```
40 5 12 50 formula .
75
```

the first thing that happens is that the values are taken from the stack and put in a special area of memory where they are tied to the names in the curly brackets in the same order as they were put on the stack. Once that happens, their order is unimportant. Their names are used to fill in the values places in the calculation.

But note that the "solution" parameter is actually a comment—anything between the -- and the } is treated as a comment. You should use this comment area to indicate what your definition leaves on the stack, exactly as in a normal stack comment.

It is important to bear in mind that the names and values you assign to named input parameters are valid only within their own colon definition. You could use the same names with the same or different values in other colon definitions without any interference.

Named input parameters become very powerful in the way you can adjust their values in the course of a colon definition. Consider, for example, this formula:

$$a^2 + b^2$$

Since the computer can compute only one square operation at a time, it needs to hold the result of one square while it calculates the second before it can add the two squares. A Mops definition for this formula would be:

```
: formula1  { a  b -- solution }
      a  a  *  -> a
      b  b  *
      a  +    .  cr  ;
```

The arrow operation (->) stores the value currently on the stack (the result of a-squared) into the named parameter, a. This overwrites the original value in a, which came from the stack in the opening instant of this definition's execution. Near the end of execution, a is recalled to be added to the results of b times b. To do the same formula without named input parameters would require several stack manipulations that sometimes trip up even the pros.

Incidentally, there are other operations you can perform to a number stored in a named input parameter. You can add a number to what is there, or subtract a number from what is there, with the ++> and --> operations. For example,

```
10 ++> denom
```

inside a colon definition adds ten to the value stored in the named input parameter named denom.

Local variables

While we're at it, we'll also introduce you to a similar concept, called local variables. They, too, appear inside curly brackets within a colon definition, but instead let you assign names to intermediate results that can occur inside such a definition. Local variables are preceded by a backslash. Take, for instance, the formula,

```
(a+b-3c)/(b+2c)
```

The formula definition would be:

```
: formula2  { a  b  c  \  num den -- result }
      a b +  3 c *  -  -> num
      b  2 c *  +      -> den

num  den  /  ;
```

In this example, a, b, and c in the curly brackets are named parameters that take on the values in the stack. The backslash indicates that the names to the right are local variables that will be called into action within the definition. In the example, the numerator and denominator are calculated separately and stored (->) in their respective local variables. Then, the local variables are recalled in the proper order for the division operation to reach the result.

An important thing to remember is that local variables aren't initialized to any particular value at the beginning of the definition. Don't assume they're initially zero, let's say. They might have anything at all in them, and it might be different in different runs of your program. Thus your first use of a local variable should be to store something into it with ->.

End of lesson 9

Lesson 10

Additional math

This is a good time to learn several other Mops math operations. They're rather simple, so you may as well get them out of the way now, and use them as you go along. We won't be saying too much about them here, but experiment with each of them for a bit to get a feeling for how they work.

One group of operations compares the values of the two topmost items in the parameters stack. The result of the comparison is placed on the stack. Here they are:

MIN	(n1 n2 -- n-min)	Leaves the smaller of n1 and n2 on the stack.
MAX	(n1 n2 -- n-max)	Leaves the larger of n1 and n2 on the stack.

The next group manipulates the signs of integers—positive or negative. One returns the absolute value (positive value) of the topmost number in the stack. The other changes the sign of the topmost number in the stack: if the original is positive, the operation changes it to negative, and vice versa. Here are these two operations:

ABS	(n -- |n|)	Leaves the absolute value of n on the stack.
NEGATE	(n -- -n)	Changes the sign of the topmost number on the stack.

Next is a laundry list of simple arithmetic shortcuts. Their meanings should be self-evident.

1+	(n -- n+1)	Adds 1 to the number on the stack.
1-	(n -- n-1)	Subtracts 1 from the number on the stack.
2+	(n -- n+2)	Adds 2 to the number on the stack.
2-	(n -- n-2)	Subtracts 2 from the number on the stack.
2*	(n -- 2n)	Multiplies the number on the stack by 2.
2/	(n -- n/2)	Divides the number on the stack by 2.
4*	(n -- 2n)	Multiplies the number on the stack by 4.
4/	(n -- n/2)	Divides the number on the stack by 4.

Displaying text

Many times in a program, you want to display text on the screen. It may be to display a heading on a screen or to "humanize" a purely numeric answer by describing what the number is. In the latter case, you are actually combining the display of a pre-planned text message with a numeric answer, which can change from execution to execution.

In Mops, the simplest way to display a text message is by preceding it with a special print command called the dot-quote, or ." in Mops notation. It's just like the dot command, but instead of looking to the stack for something to display, the dot-quote command displays everything that follows it, up to a closing quotation mark.

The quotation marks fall into a broad category of symbols in computer languages called delimiters, because they delimit or set the limits for something—in this case a text message. The text within the delimiters is called a text string, or just string. Note that for normal Mops words, spaces or tabs or carriage returns are delimiters. However for message strings we usually want to be able to include spaces as part of the string, so we use " as a delimiter instead. However, since ." is a Mops word, it must itself be delimited by a space. This space is not included as part of the string, but the first character after the space is the first character of the string.

Text strings can be made part of Mops word definitions very easily. In the following example, you'll define the word "hi" so that it prints a greeting message from the computer.

```
: hi  ." hello, this is Mops operating on the Macintosh."  cr ;
```

Now, when you type "hi" at a Mops prompt, the message between the quotes appears on the screen. Note again, that the space immediately after the ." is not part of the message, but just serves to delimit ." as a word. If the space wasn't there, Mops would try to interpret ."hello, as a word, which certainly isn't what we want.

One of the nice things about Mops is that you can use previously defined words inside the definitions of new words. Therefore, you could take the "hi" Mops word and incorporate it inside yet another Mops definition. For example:

```
: greeting   hi ." How are you?" cr ;
```

produces not only the message of "hi", but an additional text string whenever you type "greeting" at a Mops prompt. Try it.

Now combine your knowledge of arithmetic operations and text strings to humanize your earlier arithmetic word, add. In this case, you're going to redefine add. To do this, simply type in the new definition. Mops may alert you that you have redefined the word when you press Return, depending on how Mops has been set up (we will describe this later). Here's the new definition:

```
: add  ." The sum is: " + . cr ;
```

To use the new word, issue the command at the Mops prompt like this:

```
10 20 add
The sum is:  30
```

Explicit stack manipulations

While named input parameters and local variables will disguise many stack manipulations for you, there may be occasions when the order of items in the stack requires an explicit move of some values for a particular operation. Conversely, the stack may have a number on it that you simply don't need anymore, and want to dispose of. In these cases, you can choose from a series of stack manipulation commands.

Here are three stack manipulation operators that you should keep in the back of your mind:

SWAP	(n1 n2 -- n2 n1)	Switches the order of the topmost two items in the parameters stack.
DUP	(n -- n n)	Duplicates the topmost stack item and places the new copy on top.
DROP	(n --)	Removes the topmost stack item. If another item is next in line, it then becomes the topmost item.

SWAP is used most often when two values are on the stack, but their order is wrong for a subtraction or division operation. In fact, it could have been used in a less elegant definition for the problem cited in Lesson 3,

$$\frac{5 * 12 * 50}{40}$$

By putting the divisor at the bottom of the stack (the first one in), you can perform all the multiplications and then switch the order of the two remaining numbers on the stack so they divide properly. The revised operation would be:

```
40 5 12 50 * * swap /
```

The colon definition that calculates this would be:

```
: formula  ( denom num1 num2 num3 -- solution )
     * * swap / ;
```

DUP is sometimes useful for particular arithmetic applications. An example of how DUP works is to use it to calculate the square of a number. Instead of entering two exact values onto the stack, you can enter only one, duplicate it, and then multiply the two values on the stack like this:

```
4 dup *
```

Calculating the cube of a number could be performed like this:

```
4 dup dup * *
```

Therefore, you could set up a Mops word "cubed" to perform the cube calculation:

```
: cubed  ( n -- )  dup dup * * . cr ;
```

Then you could type "3 cubed" from the Mops prompt, and the answer would appear on the screen like this:

```
3 cubed
27
```

Experiment with the other stack manipulation operators described above. Place a few numbers in the parameters stack, issue the commands, and see what happens in the stack display. If you need to, you can combine two or more stack manipulation operators in the same Mops word definition as your arithmetic needs arise.

But overall, named input parameters and local variables are generally a better way to handle numbers on the Mops stack. Tracing and debugging a program is much easier than with explicit stack manipulations. And because named parameters and local variables are more intuitive, there is less chance of making a mistake in the first place.

End of lesson 10

Lesson 11

How Mops makes decisions

A decision—both the human and computer kind—is little more than the result of a test of conditions. For example: if it is true that the light switch is ON when you leave the room, then you make a small detour to hit the switch on your way out. In other words, you are testing for a certain condition in the course of your normal operation. If the condition is true, then you do something accordingly. If the condition is false, then you carry on with your normal operation as if nothing had happened.

This IF...THEN decision construction is precisely what goes on inside the computer when your program needs to test for a specific condition—like whether a number is odd or even; whether the program user typed in the correct answer; and so on.

In Mops (as in other Forths) the IF...THEN decision process is a bit different from some other languages you may know, largely because of the stack orientation. The formal description of the IF...THEN construction is as follows:

```
( n --  )   IF  xx   THEN   zz
```

If n is non-zero (true), statement xx is executed, followed by zz; if n is zero (false) the program continues with statement zz.

The IF part of the Mops decision process tests for the presence of a zero or non-zero (i.e., any number but zero) on the top of the parameter stack prior to the IF statement. Whenever the IF statement finds a non-zero number on the stack, it performs the operation written immediately following IF. From there it goes on to perform whatever operation after THEN. Whenever the IF statement encounters a zero on the stack, it performs the operation written after the THEN statement. In Mops the "THEN" means to proceed with the program after the test, as in "first do this, then do that."

You won't be able to experiment with the IF...THEN construction quite as easily as the operations you learned so far. That's because this construction must be compiled before it will run on Mops. So you'll need to put the IF...THEN statement inside a colon definition and compile it before you can run it. So type the following:

```
: test
    IF ." There is a non-zero number on the stack."
   THEN  cr ;
```

Note that since you have commenced a colon definition, when you type <ENTER> at the end of each line, that line is compiled by Mops. Alternatively, you could type <RETURN> at the end of each line, which would just enter the text into the Mops window without doing anything with it, and then you could compile the whole of the definition at once by selecting it and hitting <ENTER> (as we saw in Lesson 6).

This defines "test" as a word that performs a check on the top number on the stack. If the number is non-zero, then the statement to that effect shows on the screen. If the top of the stack contains a zero, then the statement does not appear. Try it by placing various numbers—including zero—on the stack and typing "test." Remember that an empty stack contains no numbers, and the IF operation will cause the "empty stack" error message to appear. A zero, on the other hand, is indeed a number, and it occupies space on the stack.

Two alternatives

Some decisions, however, are more complex because they involve two possible alternatives before proceeding. Take, for example, one of the most difficult decisions: getting up for work in the morning. After the alarm has gone off, and you lie in bed deciding whether you should really get going, or grab another half hour, your mind is testing certain conditions. IF you get up now, THEN you'll be on time for work, or ELSE you'll risk losing your job. IF you get up now, THEN you can get all the hot water, or ELSE you'll have to rush through the shower to get the few drops that are left after the rest of the family has showered.

This kind of decision construction has been included in Mops. Its definition is:

```
( n -- )  IF   xx   ELSE   yy   THEN    zz
```

If n is non-zero (true), xx statement is executed, followed by zz; if n is zero (false), yy is executed, followed by zz.

As with the IF...THEN construction, this decision process looks first to see if the number on the top of the stack is zero or not before it makes any decision. Now redefine "test" so it takes into account the ELSE provision:

```
: test
      IF   ." Non-zero number on stack "
      ELSE ." Zero on stack "
      THEN  cr ;
```

Place three numbers—one, zero, and three—in the stack and perform three tests:

```
1 0 3
test
Non-zero number on stack
test
Zero on stack
test
Non-zero number on stack
```

As with nearly all Mops operations, the IF operation takes the top number off the stack when it performs its check. If you will need that number for a subsequent operation, then first convert the number to a named input parameter or local variable to preserve the value for a later calculation.

Truths, falsehoods, and comparisons

You may be wondering how the IF...THEN construction can be useful if it can only determine whether or not the number on the stack is zero. You might think that this kind of test would be rather limiting in light of the "real-world" decisions that a program may have to make, such as whether two integers are equal to each other, whether one is larger than the other, or whether a number is positive or negative. Actually, the IF...THEN construction frequently operates at the tail end of a fuller decision procedure that makes the real-world decisions possible. The first part of the procedure consists of one or more comparison operators whose results are either a zero or non-zero, depending on the outcome of the comparison.

To simplify the zero and non-zero terminology, Mops adheres to a programming language convention revolving around the terms TRUE and FALSE. These words are Mops words, and represent the values that appear in the stack as a result of the comparison operations. False represents a zero in the stack; true represents any non-zero number in the stack, including negative numbers. The Mops word TRUE returns a non-zero number, of course —it returns a number which is all ones. As we'll see a bit later, this corresponds to the value -1.

Type these words now:

```
true false
```

You"ll see from the stack display that false is the same as zero, and true is -1.

Since these words—or rather the numbers they represent—are actually symbolic of a condition that has just been tested, they are sometimes referred to as flags. Flags in programs are something like markers planted in key places that symbolize a certain condition. A "true" flag signifies that a non-zero number is on the stack; a "false" flag signifies that a zero is on the stack. Another term that is used is boolean—this really means the same as "flag".

To help ingrain this true/false difference in your mind, redefine "test" so that it reinforces the way the IF...THEN...ELSE construction responds to TRUE and FALSE flags existing in the stack.

```
: test
    IF ." True "
    ELSE ." False "
    THEN  cr  ;
```

Now, place the numbers zero and four on the stack (and leave the true and false underneath them, which you put there before). Then run the test five times:

```
0 4
test
True
test
False
test
False
test
True
```

Below is a list of comparison operations that test the values of one or more numbers on the stack and leave either true or false flags on the stack. It is these operations you perform on real-world integers before performing decision operations like IF...THEN...ELSE. A new term appears in the stack notations below: boolean. This means that the result is either TRUE or FALSE flag on the stack ("boolean" is named after George Boole, who developed a logic system based on TRUE and FALSE values).

0<	(n -- boolean)	Leaves TRUE (-1) on the stack if n is less than zero; otherwise, leaves FALSE (0).
0=	(n -- boolean)	Leaves TRUE on the stack if n equals zero; otherwise, leaves FALSE.
0<>	(n -- boolean)	Leaves FALSE on the stack if n equals zero; otherwise, leaves TRUE.
0>	(n -- boolean)	Leaves TRUE on the stack if n is greater than zero; otherwise, leaves FALSE.
<	(n1 n2 -- boolean)	Leaves TRUE on the stack if n1 is less than n2; otherwise, leaves FALSE.
<=	(n1 n2 -- boolean)	Leaves TRUE on the stack if n1 is less than or equal to n2; otherwise, leaves FALSE.
<>	(n1 n2 -- boolean)	Leaves TRUE on the stack if n1 does not equal n2; otherwise, leaves FALSE.
=	(n1 n2 -- boolean)	Leaves TRUE on the stack if n1 equals n2; otherwise, leaves FALSE.
>	(n1 n2 -- boolean)	Leaves TRUE on the stack if n1 is greater than n2: otherwise, leaves FALSE.

>= (n1 n2 -- boolean) Leaves TRUE on the stack if n1 is greater than or equal to n2; otherwise, leaves FALSE.

All the math in these comparison operations should be familiar to you. Remember that these operations, like the simple arithmetic ones, are set up in postfix notation. To remember which order to put numbers on the stack, simply reconstruct in your mind how the formula would look in algebraic notation. For example, to find out if n1 is greater than n2, the algebraic test would be:

```
n1 > n2
```

In Mops, you simply move the operation sign to the right:

```
n1 n2 >
```

But in this case, Mops is testing the validity of the statement. While the numbers are tested, each is taken from the stack. If the statement is true, then a TRUE flag goes to the stack; otherwise, a FALSE flag goes there. Then an IF...THEN or IF...THEN...ELSE decision can be made on the number(s) in question.

Nested decisions

It is also possible to have more than one IF...THEN...ELSE decision working at one time. To accomplish this, you can place IF...THEN...ELSE decisions inside one another. For example, you can set up a series of decision operations that will examine a number in the stack, test it for several conditions, and then announce on the screen what condition that number meets. To do this, you'll nest several IF...THEN statements inside one another:

```
: iftest  { n --  }
      n 0<
      IF    ." less than "
      ELSE  n  0>
            IF ." greater than "
            THEN
      THEN
      ." zero "  cr ;
```

"Iftest" is defined to check whether a number is positive, negative, or zero. Enter a number in the stack and then perform an "iftest" of it. Try positive and negative numbers and zero. The number is assigned to a named input parameter (n) because it might have to be tested by both IF statements—the first IF would remove the number from the stack, leaving nothing for the second IF to test. The number is then tested whether it is less than zero. If so, "less than zero" is displayed, because the program jumps ahead to the second THEN. If the number is not negative, it is next compared to see if it is greater than zero in the second, nested IF...THEN construction. If the number is greater than zero, then the TRUE flag is noted by the second IF statement, and "greater than zero" is displayed. If the number (which has already proven to be not less than zero) is not greater than zero, then it must be zero, and only "zero" is displayed on the screen.

The key point to remember in nested IF...THEN constructions is that every IF must have a corresponding THEN somewhere in the same colon definition. They are nested much in the same way that parenthetical delimiters in math formulas are nested:

```
(a/(a-(b*c))+c)
IF    xx
      IF     ww
            IF    uu
            ELSE  zz
            THEN
      THEN   qq
THEN    yy
```

Each THEN matches the IF with which is lined up. Formatting your code this way, with corresponding IFs, ELSEs and THENs lining up, is a good idea for readability.

Logical operators

There will probably be occasions in future programs in which you will have performed two comparison operations, and the resulting flags from those operations will be sitting on top of the stack. How the program proceeds from there depends on the state of those two flags. If one flag is true and the other false, they may meet the prerequisite that only one of the comparisons needs to be true for a certain operation to take place (e.g., n1 is less than n2, but n1 is not less than zero). Conversely, you may need both flags to be TRUE for a certain operation to take place (n1 is both less than n2 and less than zero). In these special cases, you can use the logical operators, AND and OR, which we'll now describe.

Both of these operations look at the binary makeup of two numbers and produce a result. For AND, the result will have a 1 in each position where both the first and the second numbers have a 1. For OR, the result will have a 1 in each position where the first or the second numbers (or both) have a 1. For example:

```
        0001   (binary number 1)
AND     0011   (binary number 3)
        0001   (1 "AND" 3 equals 1)

        0001   (binary number 1)
OR      0011   (binary number 3)
        0011   (1 "OR" 3 equals 3)
```

The AND operation above returns a 1 for the rightmost column of bits in the binary numbers because both bits are 1. The OR operation above returns a 1 for the two rightmost column of bits in the binary numbers because one or both bits in each column are 1. The names for these operations, AND and OR, are sometimes used as verbs, as in "I want to AND 1 and 3."

In Mops, these words have the following definitions:

AND	(n1 n2 -- n3)	Performs a bit-wise AND of n1 and n2 and leaves the result on the stack.
OR	(n1 n2 -- n3)	Performs a bit-wise OR of n1 and n2 and leaves the result on the stack.

As indicated by the Mops stack notation above, the proper format for these logical operations is to place the numbers on the stack and then issue the operation name. For example:

```
1 3 AND . cr
1
```

Experiment with AND and OR in this fashion. Remember that these operations are working on the binary equivalent of the decimal numbers you type into the stack. If you have difficulty understanding an answer, try working out the problem on paper by converting each number to binary and then performing the AND or OR arithmetic on the numbers as shown above. Once you understand the concept, you can trust Mops to do these operations correctly for you at all times.

#####something here about and and or with flags.

The CASE decision

It's not uncommon to have an instance in a program in which the next step could be one of several, depending on the actual number on the stack—not just whether it's true or false. For example, a program may ask you to type a number from zero to nine. For most of the numbers, the subsequent step is the same, but for numbers 2, 6, and 7, the outcome is different. In other words, if it is the case of a "2" on the stack, then a unique operation takes place. Sure, you could run a series of comparison operations and nested IF...THEN constructions on the number to narrow it down (e.g., testing if the number is not less than two nor greater than two), but that gets cumbersome when you're testing for many numbers.

Mops's shortcut for this multiple decision making is the CASE structure. Using the example above, you could define a word like this:

```
: CaseTest  ( n --  )  ( Print TWO, SIX, SEVEN, OTHER )
     CASE
          2     OF    " TWO"      ENDOF
          6     OF    ." SIX"     ENDOF
          7     OF    ." SEVEN"   ENDOF
          ." OTHER "
      ENDCASE   ;
```

This word takes the number on the stack and checks whether it is a CASE OF 2, 6, or 7. If a particular CASE is valid, then the branch executes statements until it encounters an ENDOF delimiter. At that point, execution jumps to ENDCASE, ignoring all other statements. If none of the cases are valid, then execution continues toward the ENDCASE delimiter. If a statement is inserted before ENDCASE (as is ." OTHER " in the example), then it is executed whenever the test of cases fails. This statement is also known as the default statement, since it's the statement which gets executed by default if nothing else does.

Note particularly, that the CASE test retains the test value on the stack, and it is dropped at the end by the ENDCASE. In the default statement, particularly, you might want to make use of the test value. But if you're going to take it off, remember to DUP it or put a dummy value on the stack to be dropped by the ENDCASE.

End of lesson 11

Lesson 12

Loops

Computer programs frequently need certain operations to be repeated a specified number of times. For example, finding the sum of 10 numbers in the stack would normally take a stream of nine + statements. To a programmer's way of thinking, this makes the program several steps longer than necessary. A programmer would rather find a shortcut way of repeating that operation as many times as is needed to do the job, without increasing program size with a long series of identical commands. That's where the loop comes in.

A loop sets up a kind of merry-go-round for your program, with a beginning and an end. At the end of the loop is an instruction that tells the program to "loop back" to the beginning of the loop. All the statements between the two are repeated in their entirety each time program execution goes through the loop.

Mops has two major categories of loops: definite and indefinite. As their names imply, each category has a different way of figuring out when to stop going around in loops. The definite loop performs only as many loops as the program specifies; an indefinite loop, on the other hand, keeps looping until a certain condition is met. Let's look at each kind of loop more closely.

Definite loops

Consider the 10-number addition problem noted above. Since you know ahead of time that there will be exactly ten numbers on the stack before any addition takes place, you could use a definite loop to perform nine addition operations on the stack.

The construction of a definite loop in Mops consists of a DO...LOOP statement, which expects to find two numbers on the stack before the DO executes. The two numbers represent the beginning and ending count of repetitions that the DO...LOOP statement is to make.

Because loops work in compiled statements only, put them inside colon definitions to see how they work. Define a new word that adds up 10 numbers from the stack by performing nine repetitions of addition:

```
: addten    ( n1...n10 -- sum )
      9 0  DO  +  LOOP .  cr  ;
```

During execution, DO...LOOP counts up from zero to nine each time through the loop. After the ninth time around, the program is let out of the loop; it proceeds to display the contents of the stack (the sum) and to send a carriage return to the screen.

You may be wondering where Mops keeps track of the loop counter if the parameter stack is used to hold all the numbers that get added. The answer holds one of Mops's powerful features, called indexing, which will play an increasingly important role the more you learn about Mops.

When you typed the 9 and the 0 prior to the DO...LOOP construction in the example above, what you couldn't see was that the two numbers were automatically shifted over to another part of memory. The first number you typed (the 9) is called the limit, because that number represents the limit of how many times the loop is to be executed. The second number (the 0) is called the index. This number counts up by one each time through the loop. So, the first time the DO...LOOP construction is encountered in the above example, the index number counts up to a one; the next time to a two, and so on. When the index and limit numbers are equal, then the DO...LOOP construction "knows" that it's time to move on.

What's interesting about this kind of indexing is that you can use the index number as a counter while executing a loop. By setting the limit and index numbers to integers you need to operate with inside a loop (they can be any integers you want), you can copy the index number to the parameter stack each time around the loop and use that number for a calculation, a graphics plot point, a multiplication factor, or whatever. The Mops word that copies the index to the parameter stack is:

I (-- n) Copies the current index value to the parameter stack.

Remember that this word only copies the index; it does not disturb the index in any way. Here are a couple of examples to demonstrate.

Define a word, fivecount, that displays a series of numbers from 101 to 105:

```
    (  --  )
: fivecount  ( -- )
     106 101  DO  i  .   LOOP  ;
```

Notice that the limit is set to 106. That's because the index is incremented when execution reaches LOOP. The first time through, the index was 101, and the "I" word copied the index to the parameter stack; the dot command then displayed it on the screen. On the fifth execution, 105 was the index. When execution reached LOOP, the index incremented to 106, at which point it equalled the limit and broke out of the loop.

You can similarly use the index number to perform operations on a number passed to the parameter stack prior to execution. Consider the following definition:

```
: timestables  { n1 -- }
     13 1  DO  n1 i *  .  LOOP  cr  ;
```

If you then type "5 timestables," the program goes through twelve loops of multiplying 5 times the incrementing index number, one through twelve.

You have the flexibility in Mops to place all kinds of other statements within a DO...LOOP construction, including all those conditional decision makers covered earlier.

There will be times when you'll want to use a DO...LOOP for the sake of compactness, but the increment you wish to go by is something other than the one automatically performed by the loop (increment by 1). For those occasions, you have the optional loop ending, +LOOP. Whatever number you place in front of the +LOOP ending will be the increment that the DO...LOOP uses to adjust the index. You can even use a negative number if you wish the loop to count backwards. Here's how you would use +LOOP to take care of a countdown:

```
: countdown  ( -- )
     1 10
     DO    i  . cr
     -1 +LOOP
     ." Ignition...Liftoff!"   cr  ;
```

Notice that in this case, since the program is counting backwards, the limit is zero and the index is 10. Each time through the loop, the index is incremented by a -1. Also notice that the limit value 1 gets typed by the program—when the index is counted down and becomes equal to the limit, the loop continues, and doesn't stop until the index is counted down to the limit minus 1, unlike the situation when the index is being counted up, where the loop stops when the index equals the limit. The best way to think about this, is as if there is a "fence" in between the limit and the limit-1. Whenever the index crosses the fence, in either direction, the loop stops. This will be true even if you write a program in which the increment value changes sign during the running of the loop.

Nested loops

It is also sometimes desirable to have more than one DO...LOOP routine going on simultaneously. As with IF...THEN constructions, DO...LOOP operations can be nested inside one another. All you have to remember is to supply one LOOP (or +LOOP) for each DO within the colon definition. For example, you could add a delay loop in the "countdown" definition above to make it look like the seconds are being counted down (a better way is to use the neon words PAUSE or WAIT defined in source 'Interval'). Insert:

```
600000 0 DO LOOP
```

after the dot statement inside the original DO...LOOP operation, and again after the "Ignition" line:

```
: countdown
      1 10
      DO    i   . cr
            600000 0  DO  LOOP
      -1 +LOOP
      ." Ignition"   cr
      600000 0  DO  LOOP
      ." Liftoff"  cr  ;
```

Type "countdown" and watch the seconds tick away:

```
countdown
10
9
8
7
6
5
4
3
2
1
Ignition
Liftoff
```

Of course, in a real program, you would probably take out the "600000 0 DO LOOP" and make it another definition, perhaps called "delay"—then, if you needed to change the delay value you would only have to change your code in one place. This operation is called "factoring", since it is a bit like the mathematical operation of getting common factors in numbers.

If you are in a nested loop and need access to the outer index, Mops has a predefined word that allows you to copy that number to the parameter stack, just like "I" copies the current loop index number to the stack. That word is "J."

J	(-- n) Copies to the parameter stack the index of the next outer loop of a DO...LOOP construction .

In other words, "J" looks up the index of the loop just outside the current DO...LOOP construction, and copies that number to the parameter stack. But note that if you have factored out an inner loop into another definition, you can't use J this way—you won't get the right value. J only works with nested loops within the one definition.

LEAVE

You may have a situation in which you need to bail out of a DO...LOOP before its normal completion—perhaps because of some special case situation. The word LEAVE is available for this purpose. Here's the countdown example again, appropriately modified:

```
: countdown
      1 10
```

```
      DO    i  . cr
            600000 0  DO  LOOP
            i 7 = IF  ." Aborted!!" cr  LEAVE  THEN
      -1 +LOOP  ;
countdown
10
9
8
7
Aborted!!
```

Note that we had to remove the "Ignition" and "Liftoff" messages, otherwise "Ignition" and "Liftoff" would have appeared after the countdown was aborted, which wouldn't really be what we want. We'll show a better way of handling this shortly.

Indefinite loops

An indefinite loop is another kind of loop you'll use from time to time in a Mops program. As its name implies, an indefinite loop keeps going in circles until a certain condition exists. It can go around one time or thousands of times while waiting for that condition to occur. In Mops, that condition is the presence of a TRUE flag (non-zero number) on top of the parameters stack. One kind of indefinite loop is defined as:

```
BEGIN xxx
UNTIL
```

Performs xxx operations repeatedly until a TRUE flag exists on the parameters stack.

A useful variation is:

```
BEGIN xxx
NUNTIL
```

Performs xxx operations repeatedly until a FALSE flag exists on the parameters stack.

Here's an example of how you might use a BEGIN...UNTIL construction. In this case, the indefinite loop will be waiting for you to type a lower case "a" on the keyboard. The KEY operation pauses the program until you press a key, and then it places onto the stack a standard code number (called its ASCII code—explained later) for the next character typed. If the number on the stack is 97 Decimal (the ASCII code number for the lower case a), then a -1 (TRUE flag) is placed on the stack, and the loop ends. Otherwise, a zero (FALSE flag) is placed on the stack, and execution returns to the beginning of the loop.

```
: beginTest       BEGIN key 97 =  UNTIL  ;
```

Now, type "beginTest," and type all kinds of letters on the keyboard. Until you type an "a", the program keeps going around in circles. Another indefinite loop to remember is:

```
BEGIN   xxx
WHILE  yyy
REPEAT
```

Always executes xxx each time through the loop, and executes yyy only if a nonzero number appears on the stack; loop ends when stack shows zero.

In this case, the operation after the WHILE statement may never execute if a zero exists on the stack after BEGIN's operation (xxx).

There is also a variation to this kind of loop:

```
BEGIN    xxx
NWHILE  yyy
REPEAT
```

Always executes xxx each time through the loop, and executes yyy only if a zero appears on the stack; loop ends when stack shows a nonzero value.

EXIT

This is a good place to mention another very useful operation, EXIT. This completely exits the current definition. Here's a modified version of beginTest:

```
: beginTest
      BEGIN
            key 97 =  IF  EXIT  THEN
            key 98 =
      UNTIL  ;
```

This definion will keep running until you type either an "a" (97) or a "b" (98).

You can also write:

```
: beginTest
      BEGIN
            key 97 =  IF  EXIT  THEN
            key 98 =  IF  EXIT  THEN
      AGAIN  ;
```

The word AGAIN returns straight to the BEGIN, without testing anything. Of course, if you write a BEGIN...AGAIN loop, the loop must have some other way of terminating, such as EXIT.

If you write EXIT within a DO...LOOP, you have to remember one more thing—Mops (as with any kind of Forth) keeps some extra information around during DO...LOOP, and you have to remove this information if you are going to end a DO...LOOP in some unusual way (that is, not via LOOP, +LOOP or LEAVE. The word to use is UNLOOP. We'll illustrate this with the countdown example again:

```
: countdown
      1 10
      DO    i  . cr
            600000 0  DO  LOOP
            i 7 = IF  ." Aborted!!" cr  UNLOOP  EXIT  THEN
      -1 +LOOP
      ." Ignition"   cr
      600000 0  DO  LOOP
      ." Liftoff"  cr  ;
countdown
10
9
8
7
Aborted!!
```

You'll notice that we've been able to reinstate the "Ignition" and "Liftoff" messages, but by aborting the loop via UNLOOP and EXIT we bypass them.

When you're designing loops, it is sometimes possible for an infinite loop to slip in accidentally. Try to avoid them! Double-check the stack operations of your indefinite loops to make sure that there is always at least one condition that will allow you or your program to terminate the loop. Otherwise, your program will appear to "lock up" and be unresponsive to your keyboard input. If this happens, you'll probably have to reset your Mac. This will generally be more time-consuming than double-checking your loop terminating conditions.

End of lesson 12

Lesson 13

Mops' fixed-point arithmetic

The basic version of Mops (Mops.dic) utilizes fixed-point arithmetic, also called integer arithmetic instead of floating-point arithmetic (MopsFP.dic). The primary difference between the two is that fixed-point arithmetic functions only with integers. You had a hint of that when you started experimenting with division in Mops: the answer was either an integer quotient or a quotient-plus-remainder (both of which were integers). Floating point arithmetic, on the other hand, allows you to enter numbers with digits to the right of the decimal.

Floating-point arithmetic is convenient in many instances, especially when results of operations traditionally are other than whole numbers: financial calculations, for example, which have cents to the right of the decimal. But floating-point also has some drawbacks, which should be particularly important to you as a Mops programmer.

One is that floating-point arithmetic takes up more memory in the computer, increasing the size of the Mops kernel. This is not as significant now as it was a few years ago, when memory was much more expensive.

Second, floating-point arithmetic usually takes more time to calculate than fixed-point. Depending on your Mac model, a floating-point calculation can take up to ten times as long as the same calculation operating in fixed-point.

And third, floating-point arithmetic can be less accurate than fixed point in some calculations. You cannot, for example, multiply a number by precisely one-third in floating-point arithmetic; you must multiply by 0.33333.... There will always be some error in the calculation, which can compound itself after a couple of further calculations based on this approximation of one-third. If you multiply 9 times 0.3333333, you get 2.9999997, rather than the desired result of 9 times one-third, or 3.

Many programs have no need for floating-point arithmetic at all. For this reason, the basic Mops system has only the smaller and faster fixed-point support, with floating-point available as an option for those who need it.

But fixed-point arithmetic presents a problem of its own, because you may be accustomed to dealing with numbers other than integers—numbers like pi or percentages. To accommodate such numbers, Mops requires that you use scalars, or operations that appear to convert floating-point numbers into fixed-point numbers.

Two of the most used scalars are those that are actually special-case combinations of familiar arithmetic operations:

*/	(n1 n2 n3 -- (n1*n2)/n3) Multiplies n1 times n2 and then divides that result by n3, leaving the final result on the stack.
*/MOD	(n1 n2 n3 -- (n1*n2)/n3 remainder) Same as */ but leaves both the result and the remainder on the stack.

Notice carefully the order of the items on the stack and how they are treated by the arithmetic operations, because they are not as you would expect in a regular combination of Mops arithmetic operations. But the order allows you to better visualize the process by changing the algebraic infix notation of a problem to Mops postfix notation. To multiply 100 times two-thirds:

```
    100 * 2 / 3        becomes        100 2 3 */
```

Similar operations can be used to work with percentages. Simply put a 100 in place of the n3 in the description above and the percentage figure in place of n2.

Decimal, hex, and binary arithmetic

When Mops communicates to the Macintosh's built-in routines, it often uses numbering systems other than the traditional decimal—base 10—system. The two most often used non-decimal numbering systems are the hexadecimal and binary. Each has very different characteristics.

The hexadecimal numbering system is a base-16 system. That is, instead of numbers increasing, say, from one to two digits after the "ones" digit has cycled from zero through nine, it cycles after 15 digits. To denote the digits after 9, hexadecimal notation uses the first several letters of the alphabet. Corresponding to decimal 10 is hexadecimal A; decimal 11 is hexadecimal B; and so on through hexadecimal F. Also called "hex" for short, a hexadecimal number is usually preceded by a special sign ($) so you know that $24 is hexadecimal 24 (decimal 36) instead of the decimal 24.

The binary system, at the other extreme, has only two digits, a zero and a one. This system may not seem very useful in light of decimal and hexadecimal systems, but as you get further into the Macintosh programming environment, you'll find times when binary math is absolutely essential for ease of designing elements such as cursors, text fonts, and icons.

To show you the differences among the three bases, here is a chart of the first 20 numbers in each base:

Decimal	Hexadecimal	Binary
0	0	0000 0000
1	1	0000 0001
2	2	0000 0010
3	3	0000 0011
4	4	0000 0100
5	5	0000 0101
6	6	0000 0110
7	7	0000 0111
8	8	0000 1000
9	9	0000 1001
10	A	0000 1010
11	B	0000 1011
12	C	0000 1100
13	D	0000 1101
14	E	0000 1110
15	F	0000 1111
16	10	0001 0000
17	11	0001 0001
18	12	0001 0010
19	13	0001 0011
20	14	0001 0100

You might have noticed in this list that there is a special relationship between binary and hexadecimal in that each time one place of the hexadecimal number reaches the maximum (F), four places of a binary number reach their maximum (1111). This relationship will prove more important later on.

Although the binary numbers shown in the above list are 8 bits wide (each binary digit, that is, a 0 or 1, is called

a bit), Mops actually stores numbers on the stack as 32-bit binary numbers. Therefore, even though you type the number 10 (decimal) into the stack, the number is actually stored as:

```
0000 0000 0000 0000 0000 0000 0000 1010
```

If you were to calculate how many numbers you could describe within a 32-bit binary number, it would come out to 4,294,967,296—that's over four billion: plenty big for just about every job you'll put your Mac to. But that's four billion positive numbers. How do you work with negative numbers?

Signed and unsigned numbers

The answer lies in a special technique of Mops that takes the unsigned (positive only) range of four billion and divides it into two halves, each slightly more than two billion numbers big. One half is assigned to the positive range, the other half to the negative. In other words, the range of these signed numbers is plus-or-minus 2,147,483,647.

What distinguishes a signed from an unsigned number is the way you perform operations on them. For example, if you enter a negative number onto the stack, the minus sign shows Mops that you intend to use a signed number. If, on the other hand, you were to enter the number three billion onto the stack, Mops would know that you mean it to be an unsigned number, since anything above the plus-or-minus 2 billion range can only be unsigned.

But you can force the issue if you want, and convert the designation of a number on the stack for use in arithmetic operations and display purposes.

To understand this process, imagine that you are using a tape recorder that has a digital tape counter that counts in binary. If you set the counter to 0000 0000 and start to rewind the tape, the first thing that shows up on the counter is 1111 1111, which is actually -1 with respect to zero. But if you were to fast-forward the tape, the counter's maximum number would also be 1111 1111. That high number would correspond to the 4 billion number of an unsigned number. But as a signed number, 1111 1111 represents the start of counting backwards from zero, that is, -1.

For some hands-on experience with this concept, consider first that the dot command you learned in the early sections of this manual was actually a command to display the signed number equivalent of the number on the stack. That means that it can display numbers only within the plus-or-minus 2 billion range. Prove it now by entering 3 billion (a three and 9 zeros) on the stack. Sure enough, the stack display will show:

```
Stack:  depth 1
-1294967296
```

which is a signed number equivalent, a negative number near 1 billion.

Conversely, let's enter a -1 (a signed number) onto the stack. This time, however, you want to display it as an unsigned number. To do this, you use the U. statement, which first converts the number to an unsigned number and then types it to the screen according to the following definition:

U.	(n --) Displays the number on the top of the stack as an unsigned, single-precision number.

Try this sequence, and watch what happens:

```
-1 U. cr
4294967295
```

Here are the other unsigned operations found in Mops:

U<	(u1 u2 -- boolean) Compares two unsigned single-precision numbers. If u1 is less than u2, then leaves TRUE on the stack; otherwise, leaves FALSE.

U> (u1 u2 -- boolean) Compares two unsigned single-precision numbers. If u1 is greater than u2, then leaves TRUE on the stack; otherwise, leaves FALSE.

One last set of numbers—ASCII

You had a preview a while back of a set of numbers called ASCII codes. These are numbers that were assigned by an industry standards group to every number, letter, and symbol on the computer keyboard, plus many control codes that computers use to communicate with each other and with peripherals, such as printers. ASCII is an acronym for American Standard Code for Information Interchange. It is this standard that allows computers to communicate so effectively over telephone lines and allows so many different computer terminals to operate with a wide variety of larger computers.

Information from the keyboard reaches the Macintosh as numbers according to this code. The computer recognizes the press of the letter "a" only as the number 97 (decimal). Because each letter and symbol has a unique number, it is possible to make comparisons of a key pressed and manipulate characters on the screen with the many number crunching tools you've already learned. If you know, for example, that all capital letters of the alphabet are numbered from 65 to 90, it is possible to create a DO...LOOP that instantly prints those letters on the screen:

```
: alphabet
      91 65 DO  i  emit  cr  LOOP  ;
```

EMIT is a Mops word that displays on the screen the character that is referenced by its ASCII number. Its definition is as follows:

EMIT (n --) Displays the character referenced by ASCII number, n.

If you want to put a particular ASCII character value on the stack, you can use the Mops word &. Try typing

```
& c
```

and you'll see that the stack display shows 99, which is the ASCII value for "c".

Other Mops words that might go along with EMIT and & are:

SPACE (--) Displays a blank space on the screen.

SPACES (n --) Displays n blank spaces on the screen.

Here's a use of SPACES in the alphabet definition to demonstrate its power:

```
: alphabet
      91 65
      DO  i  dup  64 -  spaces  emit  cr  LOOP  ;
```

It is also convenient to remember that upper and lower case letters are separated by a factor of 32 regardless of the letter. This may come in handy when you need to convert upper to lower cases or vice versa.

End of lesson 13

Lesson 14

Global constants and values

Assigning recognizable names to numbers is a convenient shortcut, as you've seen with named input parameters and local variables. But as you saw, both of those kinds of names are local—they apply only to a very limited section of the program, inside a definition or local section. But Mops also has a provision called Value for assigning readily identifiable names to numbers such that they can be used throughout a program.

Your program can contain many different values because you define each value by giving it a unique name and a number that it is to hold. You define a value like this:

```
25 value Jane
```

In other words, the value named Jane is holding the number 25. To recall a value's number, all you do is type the value name, and a copy of the number is placed on the stack. Type:

```
Jane
```

and the number 25 is placed on the stack.

A value is essentially a global version of a local variable, and responds to similar operations. To store a different number in a value, you use the store arrow, like this:

```
37 -> Jane
```

This operation writes a 37 over the original number, 25. Or you can increment or decrement the number stored in a value name with the ++> or --> operations, like this:

```
17 ++> Jane
4  --> Jane
```

This adds 17 to the 37 that is already stored there, then subtracts 4. Of course you can also do a subtraction by incrementing the value by a negative number:

```
-10 ++> Jane
```

If you want to define your values at the beginning of a program without placing specific numbers in them, you can place zeros in them all, and then store (->) numbers to them when necessary:

```
0 value Joe
0 value Nancy
     :
     :
```

Note that the initial numbers you specify for you values are set up when your program is loaded by Mops. If you restart your application without reloading it, your values will still contain whatever you last put in them, not their initial numbers.

So much for theory. Now it's time to pull together all the discussions and examples of the preceding lessons and dive into a real application. In fact, in the remaining lessons, we will dissect three programs to show you precisely how real Mops programs work.

End of lesson 14

Lesson 15

One of the best ways to learn the fine points of Mops programming is to study existing programs and then work slowly to customize them by modifying methods, defining new subclasses, creating new Mops words and objects, and sending messages to the various objects in memory.

In the next few lessons, you'll be studying two programs whose source files are in the Demo classes folder as plain document files. The first one is called Sin, the second called Turtle. Although we provide a listing for you in the next pages, you might also want to print out a copy of the source code to follow along as the discussion works its way into the lesson. Sin is an excellent example of how Mops array-type data structures work. Turtle reinforces the class-object relationship.

In the source code discussions in these lessons, the code will be shown with line numbers off to the left margin. These have been inserted here only to make it easier to refer to precise lines of code when explaining various operations. There are, of course, no line numbers in Mops code.

Building a sine table

Before we proceed, it's important that you understand what these programs were designed to do—just as you should clearly define the goal and operation of every Mops program you write.

Sin will actually be a general purpose building block for a great many programs, including some you may write later. Its purpose is to create a reference table of sine values plus a fast and simple way for later program parts to retrieve sine and cosine values.

If you're a little rusty on trigonometry, a sine value of an angle is a convenient way to work with angular measurement. Mathematically, the sine of an angle is the ratio of the length of the opposite side to the length of the hypotenuse of an imaginary right triangle having that angle in it. For example, if we have an angle labeled *theta* in Figure I-14,

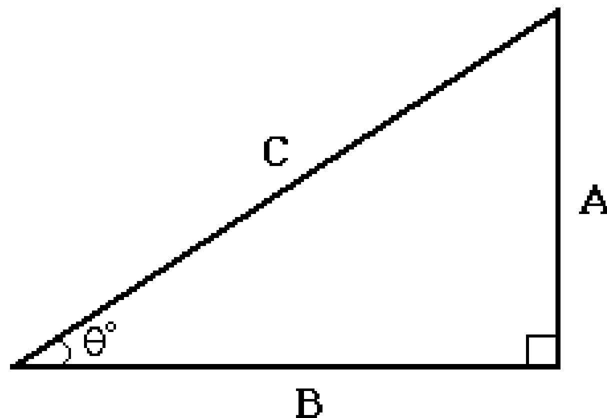


Figure 1-12.

Figure I-14

the sine of *theta* equals the length of A divided by C. If you were to calculate all possible values for sin *theta*, from 0 to 360 degrees and plot the results, you'll find that the values trend up and down throughout the circle, including two quadrants with negative values (see Figure I-15).

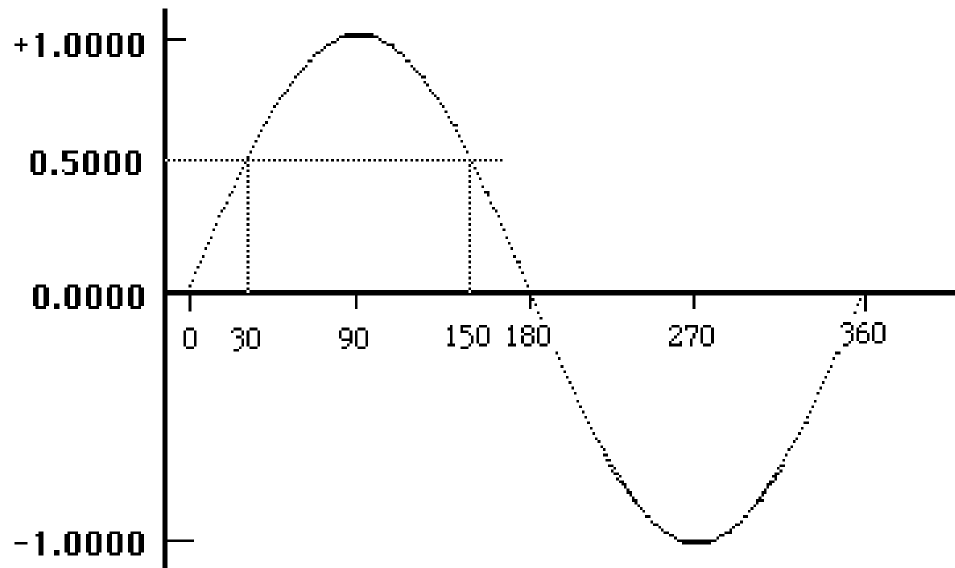


Figure I-15

Notice, however, that two angular measurements can , for example have sine values of 0.5. In the first quadrant, it's at 30 degrees. In the second quadrant, it's at 150 degrees—30 degrees from the "zero" value. In other words, the sin values in the first and second quadrants are mirror images of each other. The same is true for quadrants three and four. And the relationship between one half (0-180 degrees) to the other (180-360 degrees) is that the second half mirrors the first, but as negative values. Therefore, if you have a table of sine values for 0-90 degrees, it is a relatively simple matter to calculate the corresponding values in each of the remaining quadrants. The Sin program takes care of both the table and calculations.

Some graphics programs will likely need to fetch sine or cosine values to draw sophisticated shapes on the screen. Sin (and its classes TrigTable and Angle) will probably come in handy for you in the future.

Sin will often be summoned from the second program, Turtle. The intent of Turtle is two-fold. First of all, it will create class definitions of a pen and a polygon that you'll use to experiment developing a Logo-like environment. Turtle will also use the pen and polygons it creates (along with definitions from Sin) to draw some sophisticated graphics on the screen. As it turns out, these graphics will be incorporated into yet a third demonstration program, grDemo, which is the subject of the last lessons in this Tutorial.

This building block approach is a common tactic in designing a Mops program. Carefully, generically designed building blocks, such as Sin and parts of Turtle, can be used in a wide variety of programs, making it easier and faster to assemble programs from your library of proven blocks.

```
1     \ These classes obtain the sine and cos of an angle by table lookup.
2     \ Modified from the original Neon version by Mike Hore.
3
4     \ The main class is ANGLE, which has SIN: and COS: methods that look
5     \ up a table defined with the TRIGTABLE class.
6
7
8     need  struct1
9
10
```

```
11    :class  TRIGTABLE  super{ wArray }
12
13          4     wArray  AXISVALS        \ 90 degree values
14
15    :m SIN:  { degree \ quadrant  -- sin }
16    \ Looks up a sin * 10000 of an angle
17
18          degree 360 mod                \ Put angle in range -359 to +359
19          dup 0< IF  360 +  THEN        \ Now 0 to +359
20          90  /mod          \ Convert angle to range 0-89 and get quadrant
21          -> quadrant   -> degree
22          degree                        \ Test for an axis
23          NIF   quadrant  at: axisVals  \ If an axis, get value
24          ELSE  quadrant 1 and          \ True for "mirror" quadrants 1 and 3
25                IF    90 degree  -      \ Create mirror image
26                ELSE  degree
27                THEN
28                at: self                \ Get sin for this degree
29                quadrant 2 and          \ True for "negative" quadrants 2 and 3
30                IF  negate  THEN
31          THEN  ;m
32
33    :m  COS:    \ ( degree -- cos )
34          90 +   sin: self  ;m          \ Cos is sin shifted by 90 degrees
35
36    :m  CLASSINIT:
37          0      0 to: axisvals
38          10000  1 to: axisvals
39          0      2 to: axisvals
40          -10000 3 to: axisvals  ;m
41
42    ;class
43
44    90  TrigTable  SINES    \ system-wide table of sines
45
46    : 's        \ ( val degree -- )    Fills a Sin table entry
47          to: sines  ;
48
49    00000 00 's   00175 01 's   00349 02 's   00524 03 's   00698 04 's
50    00872 05 's   01045 06 's   01219 07 's   01392 08 's   01571 09 's
51    01736 10 's   01908 11 's   02079 12 's   02250 13 's   02419 14 's
52    02588 15 's   02756 16 's   02924 17 's   03090 18 's   03256 19 's
53    03420 20 's   03584 21 's   03746 22 's   03907 23 's   04067 24 's
54    04226 25 's   04384 26 's   04540 27 's   04695 28 's   04848 29 's
55    05000 30 's   05150 31 's   05299 32 's   05446 33 's   05592 34 's
56    05736 35 's   05878 36 's   06018 37 's   06157 38 's   06293 39 's
57    06428 40 's   06561 41 's   06691 42 's   06820 43 's   06947 44 's
58    07071 45 's   07193 46 's   07314 47 's   07431 48 's   07547 49 's
59    07660 50 's   07771 51 's   07880 52 's   07986 53 's   08090 54 's
60    08192 55 's   08290 56 's   08387 57 's   08480 58 's   08572 59 's
61    08660 60 's   08746 61 's   08829 62 's   08910 63 's   08988 64 's
62    09063 65 's   09135 66 's   09205 67 's   09272 68 's   09336 69 's
63    09397 70 's   09455 71 's   09511 72 's   09563 73 's   09613 74 's
64    09659 75 's   09703 76 's   09744 77 's   09781 78 's   09816 79 's
65    09848 80 's   09877 81 's   09903 82 's   09925 83 's   09945 84 's
66    09962 85 's   09976 86 's   09986 87 's   09994 88 's   09998 89 's
67
68    : SIN    sin: sines   ;
69    : COS    cos: sines   ;
```

```
71    :class   ANGLE  super{ int }
72
73    :m SIN:  get: self   sin  ;m
74    :m COS:  get: self   cos  ;m
75
76    ;class
```

How the sine table works

Let's start with the Sin source code, which is numbered from lines 1 to 72.

Lines 1-5

These lines are comments that serves as a plain English heading for the source code, describing what this module does, who wrote it, and what its main features are. This particular module creates a table of sine values that Turtle will use to draw complex curves and graphics. We use the "backslash" type of comment here, in which a word consisting of just a backslash causes Mops to ignore the rest of that line.

Line 8

This line causes Mops to load (compile) the file "struct1", if it is not already loaded. This file contains the definitions for the classes wArray and bArray which we will need here. The use of the syntax need <name> means that you don't have to worry about whether the file <name> is already loaded or not. In fact, it may sometimes be loaded, sometimes not, at different stages of your program development. Using the "need" syntax means that you are explicitly stating the requirements of this source file; Mops will then take care of the details.

Line 11

Here marks the beginning of a class definition for the class TrigTable. This class establishes the rules and procedures that will be followed for looking up sines in a sine table (the table is created in lines 44-66). Since the sine table will be a list of sine values in fixed-point arithmetic (in a range of 0 to 10,000), two bytes of data could be used for each entry (10,000 decimal = 2710 hex—each two-digit hex number takes up one byte of memory). Class TrigTable is defined as a subclass of class wArray.

If you look at the source code listing for the superclass wArray (in the file struct1), you'll notice that wArray is defined as an indexed class:

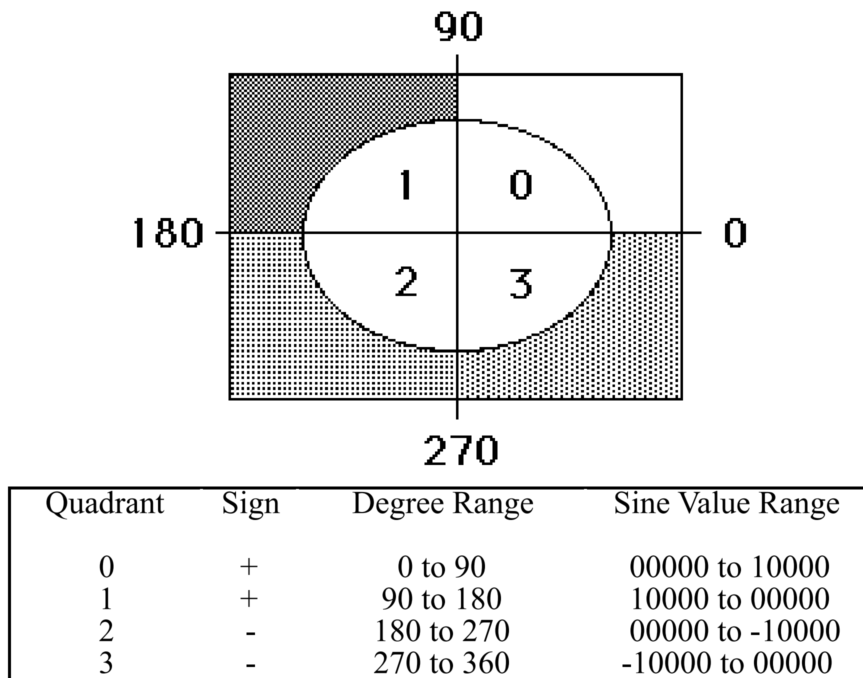
```
:class      WARRAY  super{ indexed-obj }  2 indexed
```

When a class in indexed, it means that every object created of that class must explicitly state how big an area of memory is to be reserved for its private data—how many data slots should be reserved. The number 2 in the class wArray definition indicates that each slot is to be 2 bytes wide. When it comes time to create an object from an indexed class, the line of code must begin with the number of data slots that object will need (each slot has a unique index number associated with it). In line 44, for example, the object Sines created of class TrigTable is reserving 90 slots; each slot is 2 bytes wide because TrigTable inherits wArray's 2 byte wide indexed class behavior. Indexing should become more clear as we describe the rest of this class definition, and see some practical examples.

Line 13

This line establishes the instance variable (ivar) for an object of class TrigTable. Every object created from class TrigTable will have space reserved for the array created here, as well as the indexed data noted above. The array is preceded by the number of elements that it will contain in every instance of class TrigTable, 4 in this case.

This array, AxisVals, is a 4 element array of 2 byte cells. The range of values to be stored in this array is from -10,000 to +10,000 (the integer values the program will use to signify sine values). The values in these four cells will be the sine values (times 10,000) of the 90 degree multiples (0, 90, 180, and 270 degrees), and will play a role in the calculation of the sine value later in this class definition. See Figure I-16 for a summary of the four quadrants, their signs, and sine values.



Quadrant	Sign	Degree Range	Sine Value Range
0	+	0 to 90	00000 to 10000
1	+	90 to 180	10000 to 00000
2	-	180 to 270	00000 to -10000
3	-	270 to 360	-10000 to 00000

Figure I-16

Line 15

This begins the definition of the method SIN:. The notation { degree \ quadrant -- sin } indicates that there is one named input parameter for this method, called "degree", and one local variable, "quadrant". References to such values are made by their name, not by their stack location, thus eliminating much stack manipulation in the course of calculating sine values in the next several lines of code. Within the definition, "quadrant" will be used to store the value of the quadrant (0, 1, 2, 3) for which the sine is being calculated.

This curly braces notation also doubles to give the stack effects of the execution of the method. This tells you that if you use this SIN: method as a selector in a message, and if you pass a degree figure as a parameter, (e.g., 90 SIN: object), then the corresponding sine value would be left on the stack when the method's computations are completed.

Line 16

The comment tells you what is happening in this method: the program looks up the sine value of an angle (in degrees). In the calculations the actual sine values will be multiplied by a factor of 10,000. All sine values in the sine table, therefore, will be integers.

Lines 18-34

Next comes the actual calculation and retrieval of the sine values. Because the math in this calculation is so tightly interwoven with IF...THEN decision constructions, we will trace what happens to the stack at each step, as well as explain why various operations are performed.

As an overview, we can say that the math calculations first convert the degree value to be in the range 0 - 359. Allowance is made for degree values entered as negative numbers, or degrees of magnitude 360 or greater. Once the degree is thus normalized, it is converted to the equivalent degree in the range 0 - 89 and the quadrant is saved for doing mirror image calculations and determining the sign. For degrees on an axis (0, 90, 180, or 270) the sine is gotten from the ivar AxisVals, since this is the simplest way to handle these special cases. Otherwise a lookup is performed on the TrigTable array.

To best understand the operation of the decision processes in this section, we will follow what happens to the values on the stack when we try degree values less than 90 degrees, exactly 180 degrees, and a value in the third quadrant. But to do this properly, we should go on to explain how the arrays are filled with the values that the method SIN: will be retrieving, and what those values mean.

Lines 36-40

The method CLASSINIT: is a special method that executes whenever an object of the current class is created. The operations in this particular CLASSINIT: are four messages, all of them TO: operations. The TO: selector of these messages is defined by a TO: method in the receiver's class. Since the receiver is the ivar AxisVals, the class in question is wArray.

For arrays, the methods AT: and TO: are the equivalents of GET: and PUT: for ordinary scalar objects. They expect an index on the stack at the start, to indicate which array element is to be accessed.

Since class TrigTable has now been defined (all the code from line 10 through line 42), we can now create an actual table in memory as an object of that class. The statement in line 44 does just that, establishing an indexed array object, called Sines, capable of storing 90 values in addition to the ivars, Signs and AxisVals. At this point, no values have been entered into the 90 cells of the Sines array, but the space is there, ready for values to be plugged in. The array bears the characteristics of arrays defined in TrigTable's superclass, wArray.

Lines 46-66

While the columns of numbers in lines 49 through 66 may look intimidating, they are really nothing more than the values of what becomes a computer version of a lookup table, like the kind at the end of a trigonometry book. Lines 46 and 47 define a Mops word, 's (the apostrophe is pronounced "tick") that performs a similar kind of TO: storage operation as demonstrated in CLASSINIT:, but this time the storage is to an object called Sines, which is our TrigTable object. When Sines receives a message with a TO: selector, Sines first looks in its own class (TrigTable) for a matching definition. Since there is none here, Sines then looks to its superclass wArray, where it finds a TO: method.

The stack notation for this definition uses round parentheses rather than braces, since it is a straight comment, not a definition for named parameters or local variables. This definition makes no use of named parameters or locals. To make it quite clear, we have put a \ first, which makes the remainder of the line a comment anyway. This isn't required at all, but it just prevents confusion with the braces syntax, which doubles for a stack notation and a definition of named parameters or locals.

The table was designed so that the values of the degrees to be looked up would range from 0 to 89. That way, these very degree values will have double duty as index numbers to the respective sine values in the table. Therefore, when it comes time (in the SIN: method, above) to lookup a sine value in the table, the degree value coming in as a parameter from a message will be used as the index value associated with the desired sine value. We'll see how that works in a moment, but that's why the stack notation in line 46 indicates that the parameters to be passed with each 's operation are the sine value and the angle in degrees, when in actuality, the TO: selector sees the degree figure only as an index number.

The sine values, then, are added to the table by the long series of 's operations, each preceded by the sine value (times 10,000) and the double-duty index/degree value.

What happens on the stack

Now we can go back to method SIN: in lines 15 to 31 to see what happens when we send three different degree values and the SIN: selector to the object Sines. The three values will be 35, 180, and 293 degrees. In the listings below, the numbers next to each operation indicate the actual numbers on the stack at that instant of execution. When more than one number is one the stack, the topmost number in the listing is the number on the top of the stack.

Statement	35 degrees	180 degrees	293 degrees
degree	35	180	293
360	360	360	360
	35	180	293
mod	35	180	293
dup	35	180	293
	35	180	293
0<	0	0	0
	35	180	293
IF	35	180	293
360	:	:	:
+	:	:	:
THEN	:	:	:
90	90	90	90
	35	180	293
/mod	0	2	3
	35	0	23
-> quadrant	35	0	23
-> degree	---	---	---
degree	35	0	23
NIF	---	---	---
quadrant	:	2	:
at: axisVals	:	0	:

ELSE	---	:	---
quadrant	0	:	3
1	1 0	:	1 3
and	0	:	1
IF	---	:	---
90	:	:	90
degree	:	:	23 90
-	:	:	67
ELSE	---	:	:
degree	35	:	:
THEN	35	:	67
at: self	5736	:	9205
quadrant	0 5736	:	3 9205
2	2 0 5736	:	2 3 9205
and	0 5736	:	2 9205
IF	5736	:	9205
negate	:	:	-9205
THEN	5736	:	-9205
THEN	5736	0	-9205

Now for a description of what happens to each degree value.

The mod operation in line 18 provides the stack with the remainder of dividing the degree entry by 360. If the entry was 360 or more, this will normalize the degree value to be between 0 and 359. If the entry was negative the mod operation returns a negative value between -359 and 0, and further normalization is required. Line 19 tests the result to see if it was negative, and if it was negative, adds 360 to convert it to the equivalent positive angle, correctly in the range 0 to 359.

The /mod operation on line 20 takes the normalized degree value off the stack and returns a quotient and remainder. A quotient of zero indicates it is in the upper right quadrant, a one places the degree in the second quadrant, and so on. The remainder becomes the degree value that will be checked against the sine table, since the table contains values for only a 90 –degree chunk of the full 360 degree range. On line 21 these values are taken from the stack and put into local storage.

For the next operation on line 22, we recall the value from "degree" (but this does not remove it from "degree," it only copies it onto the stack) and test to see if it is equal to zero.

If the value is zero, that means that the degree value is a multiple of 90 degrees, and therefore lies on a boundary between two quadrants. To save time and calculation, the sine values for those four boundaries have been stored in the AxisVals array. Since the degree value is zero, the operation after the NIF statement on line 23 is performed. The quadrant value saved earlier is placed on the stack and used as an index for the AT: selector. The AT: method in AxisVals' class, wArray, is the opposite of the TO: storage operator, which was used to place values in the arrays. The AT: operation instead <u>fetches</u> a value from an array object (in this case named AxisVals) according to the index number that is on the top of the stack. In our 180 degree example, a value of 2 was saved in quadrant and the put on the stack. The value in the AxisVals cell corresponding to the index "2" is then placed on the stack (it has only been copied from the array, not removed). At this point, the final sine value is in the stack, so there is no further operation needed. Following the rules of nested IF...ELSE...THEN statements, execution continues to the outermost THEN statement, which is at the end of the method.

But when the degree value is not zero, much more happens. The quadrant value is ANDed with 1 on line 24 and tested to see if is 1 or 3. If so, then the degree value is recalled and has 90 degrees subtracted from it on line 25 (sine values increase to 90 degrees, then decrease to 180 in a reverse, mirror image). Otherwise, just the degree value is placed on the stack again on line 26.

In line 28, the AT: selector takes the index value currently on the stack (it also happens to be the degree to be checked in the sine table) and fetches the value from the Sines array. The "self" notation tells Mops to perform the AT: fetch on the Sines object.

That AT: fetch operation places the sine value from the table on the stack. One last job remains—to determine if the sine value is positive or negative. The quadrant number is ANDed with 2 on line 29. If the quadrant is 2 or 3, which are the quadrants for which the sine is negative, the result of this AND will be non-zero. In this case the sine value, which is all that remains on the stack, is made negative (with the negate operation of line 30), otherwise, it stands positive, and the method ends.

The COS: method in lines 33 and 34 uses the power of the SIN: method, but simply modifies it to take into account the mathematical relationship between a sine and cosine of an angle. A cosine can be calculated from a sine by phase shifting 90 degrees.

At this point in the program (up to line 66), the kind of message you would send to calculate the sine of a degree value would be:

```
125 sin: Sines
```

To simplify this even more, two Mops definitions are added (lines 68-69). Each word sends a message like the one above. Thereafter, the only code you need in a program to obtain the sine of an angle is:

```
125 sin
```

Lines 71 - 76

class Angle provides an example of how the sin and cos definitions in lines 68 and 69 can be used in other class definitions, even though those words are defined by messages to objects of a different class. This class, an integer class, has two methods, SIN: and COS:. They may appear to have the same method names as methods in class TrigTable, but there will be no interference between the two. That's because if you create an object of class Angle, that object looks up methods only in its own class hierarchy. It doesn't even know class TrigTable exists. When a method in class Angle uses the new Mops word "sin," it lets the word reach into memory to do what it has to, even if it means working in other classes—all without disturbing the integrity of class Angle.

The "get: self" message (lines 73 and 74) retrieves the value of the integer stored in an object created from class Angle. To store a value in that object, you would need to look through class Angle's hierarchy, until you found a PUT: method in the Int superclass that stores the value. For example, if you create an object

```
angle Narrow
```

you are setting aside a cell in Narrow's memory for an integer, because class Angle is a subclass of the Int class. You would then need to send the message:

```
30 put: Narrow
```

to store the value, 30, in the object Narrow. After that, you can send the message

```
sin: Narrow
```

which sets the SIN: method in class Angle to work. The value, 30, is retrieved by the get: self operation, and then the sine value is calculated by the Mops word, sin. With Mops.dic loaded into memory, try this out yourself. Create an object of class Angle. put: a value in the object. Then send messages to that object to calculate the sine and cosine of the value.

End of lesson 15

Lesson 16

Building a turtle graphics program

Now we can look at a graphics program, called Turtle. It defines a number of complex graphics curves and a way you'll be able to create a mini-Logo language out of several definitions in the program. We'll have the first involvement with Macintosh parameters and Toolbox calls.

```
1     \  Turtle Graphics Objects for Demo
2
3     need  sin
4
5     decimal
6
7     \ Class PEN defines a turtle-graphics pen.
8
9     :class      PEN   super{ object }
10
11    record{           \ These first 5 ivars comprise a PenState structure
12          point PnLoc             \ location of pen
13          point PnSize            \ width,  height
14          int   PnMode
15          var   PnPatLo
16          var   PnPatHi
17    }
18          angle Direction
19          point homeLoc
20          int   maxReps
21          int   initLen
22          int   deltaLen          \ change in len
23          int   deltaDeg          \ change in angle
24
25    :m GET:    ^base   call GetPenState  ;m         \ Save state here
26    :m SET:    ^base   call SetPenState  ;m         \ Restore from here
27
28    :m TURN:  ( deg -- )    +: direction  ;m
29
30    :m UP:      90 put: direction  ;m
31
32    :m MOVETO:  \ ( x y -- )      Draws a line to x,y if pen shows
33          set: self  pack  call LineTo  get: self   ;m
34
35    :m MOVE:  { dist  -- }  \ Draws dist bits in current direction
36          set: self  cos: direction  dist * 10000 /
37          sin: direction  dist * 10000 /   negate
38          pack  call Line  get: self  ;m
39
40    :m GOTO:    \ ( x y -- )      Goes to a location without drawing
41          put:  PnLoc  ;m
42
43    :m CENTER:  \ ( x y -- )      Sets the center coordinates
44          put: homeLoc  ;m
45
46    :m HOME:    \ ( -- )          Places pen in center of Mops window
47          get: homeLoc  goto: self   ;m
```

```
48
49    :m SIZE:    \ ( w h -- )      Sets size in pixels of drawing pen
50          put: PnSize  ;m
51
52    :m INIT:    \ ( x y w h mode  -- )
53          put:  PnMode  put: PnSize   put: PnLoc  ;m
54
55    :m PUTRANGE:      \ ( initlen dLen dDeg -- )    Sets parameters
56          put: deltaDeg  put: deltaLen  put: initLen  ;m
57
58    :m PUTMAX:   ( maxReps -- )    put: maxReps  ;m
59
60    :m CLASSINIT:   get: self   200 put: maxReps  ;m
61
62    :m SPIRAL:  { \ dist  degrees delta  reps -- }
63                \ Draws a spiral of line segments - Logo POLYSPI
64          home: self
65          get: initLen  -> dist  get: deltaLen  -> delta
66          get: deltaDeg  -> degrees   0 -> reps
67          BEGIN 1 ++> reps  reps  get: maxReps  <
68          WHILE dist  move: self   degrees  turn: self
69                delta ++> dist
70          REPEAT  ;m
71
72    :m DRAGON:  \ ( n -- )   Dragon curves from Martin Gardner
73          dup
74          NIF   get: deltaLen  move: self  drop
75          ELSE  dup  0>
76                IF    dup  1-  dragon: self
77                      get: deltadeg  turn: self
78                      1 swap -   dragon: self
79                ELSE
80                      -1 over -   dragon: self
81                      360 get: deltadeg  -  turn: self
82                      1+  dragon: self
83                THEN
84          THEN  ;m
85
86    :m LJ:  { \ reps -- }   \ Draws an infinite Lissajous figure
87          up: self   0 -> reps
88          get: initLen  get: direction *  cos 120 /         getX: homeLoc +
89          get: deltalen get: direction *  sin 120 / negate  getY: homeLoc +
90          goto: self
91          BEGIN   1 ++> reps  reps  get: maxReps  <
92          WHILE
93                get: initLen  get: direction *  cos  120 /  getX: homeLoc +
94                get: deltaLen get: direction *  sin  120 /  negate
95                getY: homeLoc +   moveTo: self
96                get: deltaDeg  turn: self
97          REPEAT  ;m
98    ;class
99
100   \ Define a Smalltalk Polygon object as subclass of Pen
101
102   :class  POLY  super{ pen }
103
104         int   Sides             \ # of sides in the Polygon
105         int   Length            \ Length of each side
106
```

```
107   :m DRAW:  { \ turnAngle -- }
108         360  get: sides  /   -> turnAngle
109         get: sides  0
110         DO    get: length  move: self
111               turnAngle  turn: self
112         LOOP  ;m
113
114   :m SIZE:    \ ( len  #sides -- )    Stores sides and goes to Home
115         get: self  put: sides  put: length
116         home: self   up: self  ;m
117
118   :m SPIN:  { \ reps -- } \ Spins a series of polygons around a point
119         home: self  10 get: initLen  size: self
120         0 -> reps
121         BEGIN reps  get: maxReps  <
122         WHILE draw: self  get: deltaDeg  turn: self
123               get: deltaLen  +: length  1 ++> reps
124         REPEAT  ;m
125
126   :m CLASSINIT:     \ Default Poly is 30-dot triangle
127         30 3 size: self  100 put: maxReps  ;m
128
129   ;class
130
131   \ Create a pen named Bic
132   Pen   BIC
133
134   \ Create a Polygon named Anna
135   Poly  ANNA
136   60 4  size: Anna
```

Line 3

The statement "need sin" ensures that the Sin file is loaded before this one. This file makes use of the Angle class we defined in the file Sin (described in the previous lesson).

Line 5

"decimal" ensures that all numbers to follow will be in decimal, just in case a different number base was current before. Incidentally, you can place different portions of your program in different number bases, but you may have less difficulty remembering what number base you're in if you stay in decimal and precede any hex number with a dollar sign and a space (e.g., $ AE9F).

Lines 7 - 23

Beginning on line 9 is the definition of a major class for this program, the one that defines the characteristics of a pen that draws on the Mac screen. We should point out that by defining a drawing pen in Mops's object-oriented environment, you can have more than one pen drawing object in a given section of the screen (e.g., a window). The Mac Toolbox on its own does not give you this power. Consider it an added bonus of using an object-oriented language such as Mops. As you can see in lines 13 - 23, there are many instance variables for this class. Some are points, some are integers, a couple are variables, and one is an angle as defined in the class Angle (from the previous lesson).

As the comment in line 11 indicates, the first five instance variables are the parameters that a Macintosh Toolbox call, PenState, requires. For details on what the PenState parameters are, Inside Macintosh's Quickdraw chapter is the best source. There you learn that PenState takes four variables, called pnLoc (a coordinate point), pnSize (a coordinate point indicating the number of pixels wide and high—from coordinate 0,0—the pen is), pnMode (an integer), and pnPat (an 8-byte representation of the pen pattern discussed fully in Inside Macintosh). Corresponding variables are set up in this class so that any object created from this class will have those parameters stored in the right place and in the right order.

Since this is a Toolbox structure, we have to declare it as a record, as we discussed in Lesson 5. Ivars declared within records have no extra Mops housekeeping information between them, and this is what Toolbox structures require.

The reason PnPat is divided is because the largest basic data structure readily available from the predefined data structure classes is four bytes wide: the VAR. What we can do, then, is break up the 8-byte pnPat variable into two 4-byte chunks, called PnPatLo and PnPatHi, with PnPatHi holding the leftmost byte values.

The remaining instance variables will be used for other purposes in the methods definitions of this class. If you were building this class from scratch, you would probably be inserting new instance variables in this list as you find need for them while defining methods.

Lines 25 - 26

These two methods will be used frequently whenever an object of this class draws something on the screen. The first, GET:, copies the values of the Pen State variables from the Macintosh Toolbox to the ivars of an object. It's like taking a snapshot of the parameters at a given moment. Thus, after you move the pen to point x,y, a GET: saves the PenState conditions in an object's memory space. Later, when it comes time to pick up where you left off, you can SET: the parameters, which copies them from the object's memory to the Toolbox.

With the PenState variables saved within an object's "private data," other objects can use the same Toolbox routines without destroying the parameters of the first object. For example, if you tell the Toolbox to position the class Pen object named Scripto1 at coordinate 1,1, and then save those coordinates in Scripto1's data area, you are then free to instruct the Toolbox to position Scripto2 at 100,120, without affecting the data in Scripto1. Later, when you need to work with Scripto1, the SET: command reminds the Toolbox where Scripto1's position was the last time.

Lines 28 - 60

The next twelve methods are responsible for manipulating the parameters that affect any object of this class. For example, TURN: increments the angle value stored in an object's Direction ivar (+: Direction) by the number of degrees passed to it in a message, like

```
30 turn: Scripto1
```

The Direction ivar is used by sin: and cos: methods from the last lesson. These correctly handle degree values of greater than 359 degrees, or less than 0 degrees. For this reason, TURN: does not concern itself with whether the new Direction is in the range 0 - 259 degrees.

UP: (line 30) simply places a 90 in the data cell of an object's Direction ivar. This is consistent with the notation of the last chapter where the up position is 90 degrees. This will be used in a positioning message later to reset the orientation of objects drawn with a pen object from this class.

The MOVETO: method (line 32) features a Mops word that's new to you: Pack. First of all, the stack notation indicates that this method requires two parameters for the destination coordinate. The method starts out by copying to the Toolbox (set: self) the PenState values in the object's PenSt ivars. The set: self message does not affect the stack, since all data movement is going on behind the scenes between the object's ivar space and the Toolbox. That means that both parameter integers are still on the stack after the set:

self operation. The Pack operation takes those two 16-bit integers, each of which is sitting in a 32-bit stack cell, and combines them into one stack cell in a special format.

To watch Pack in action, place two numbers on the stack, and list the stack (we'll only show the parameter stack contents here, since that's all we're concerned with now). Watch what happens to the hex values on the stack:

```
255 20 .s
Stack:
      20  $    14
     225  $    FF

pack .s
Stack:
1310975  $    1400FF
```

In other words, when you pack the top two stack entries, the top entry becomes the most significant byte(s) of the compacted entry. The only reason we need to bother with the word Pack is that the QuickDraw call, LineTo (and many others) requires dual integer parameters be passed this way. Therefore, the packed stack is ready for the next operation in this method, call LineTo, when it comes along. The LineTo QuickDraw procedure, as noted in Inside Macintosh, draws a line from the current pen location (the one set in the Toolbox by the set: self operation) to the coordinate specified in the parameters. As soon as the drawing is completed, the new pen state is saved in the object's memory (get: self).

Lines 35 - 38 present another kind of line drawing. This time the location of the destination point is determined by the length (in pixels) and the direction (as retrieved from the Direction ivar). This method uses a named input parameter, Dist, because it will be much more convenient to recall the value for each of the two calculations that will be performed on it in this method. Notice that this method makes use of the sin: and cos: methods defined in the Sin program earlier. That means that Sin must be loaded into Mops before Turtle. The statement need Sin on line 3 ensures this.

The operations in MOVE: should now be familiar to you. The current pen state is copied from the object's ivar to the Toolbox. Then the sine of the current direction (the object's Direction ivar is the source of the information) is multiplied by the distance in pixels, and then divided by 10,000 (remember, sin's values have been multiplied by 10,000 for ease of handling) to obtain the x-coordinate for the destination point (which remains on the stack). The y-coordinate is calculated by the operations in line 37. Finally, the two coordinates are packed into one cell and sent to the QuickDraw routine, Line, which draws the new line. After the drawing is completed, the pen state is saved in the object's memory (get: self).

The next four methods, GOTO:, CENTER:, HOME:, and SIZE: should be largely self-explanatory. All of them but HOME: place new values into specific ivars, including one that affects some values of the pen state. HOME: simply retrieves the most recent value stored via the CENTER: method, and moves a pen class object to that location. The values you pass to CENTER: depend on the size of the displaying window, because coordinates are relative to the upper left corner of a window, no matter where it appears on the screen. (For comparison, the smallest Mac screen is 512 pixels horizontally by 342 vertically.)

INIT: (line 52) allows an object to respecify up to three pen state parameters (mode, size, and location) by way of a single message. All parameters must be sent with the message, even if only one of them is to be changed.

Line 55's method, PUTRANGE:, places values into an object's ivar slots that will be used as parameters for some fancy graphics later in the program. The names stand for a change (delta) in degrees, a change in length, and an initial length.

PUTMAX: is the method that allows you to set a value for the maximum number of repetitions some of the graphics images should make. The effect of the parameter will become more apparent when we get to the figures themselves.

In line 60, the now familiar CLASSINIT: method is performed when an object of this class is created. It first saves a copy of the current pen state parameters (the ones the Toolbox starts up with) from the Toolbox into an object's first five ivars (get: self). Finally, the maxReps ivar for the object is set to 200.

Lines 62 - 98

In these lines are three methods that are largely Mops versions of math calculations for three types of graphics images: spirals, dragon curves, and Lissajous (pronounced Lih-sah-zhoo') figures. It's not important for our discussion here to understand the inner workings of these graphic routines. You can, of course, trace the processes yourself, if you like.

We do, however, want to call your attention to the application of local variables in SPIRAL: (and in LJ:). The backslash inside the curly brackets signifies that these names are local variables, rather than named input parameters (see MOVE: above). As noted in an earlier lesson, the local variable names are used strictly inside a definition, and have no relation to named input parameters in the same definition.

In line 62, SPIRAL: declares four local variable names: dist, degrees, delta and reps. In line 64, the pen is moved to the center of the current drawing window. Dist and delta are given values in line 65 by first fetching values from two of the object's ivars, initLen and deltaLen, and then storing the values in their respective local variables (via -> operations). The third local variable, degrees, gets its value in line 66 after the deltaDeg ivar value is fetched from the object's memory. Reps is initialized to zero, and will be used as a counter to compare to maxReps. Once these local variables have values stored in them, they can be used throughout that method for whatever calculations are desired, as shown in the BEGIN...WHILE...REPEAT structure in lines 67 - 70. Without local variables, you would have to arrange for a significant amount of stack manipulation to keep the right values in the right places for calculation. It also simplifies your job of converting complex formulas into Mops, since you can construct your methods using familiar value names in your operations.

This means, of course, that the program will have to load values into initLen, deltaLen, and deltaDeg before a SPIRAL: selector message can be sent. But that's why PUTRANGE: was defined earlier.

class Pen ends with the ;class delimiter on line 98.

Lines 100 - 129

The next section is another class definition. This class, Poly, is a subclass of Pen, so it inherits the methods and ivars of Pen. Therefore, if you create an object of the class Poly, you can still issue messages with selectors like MOVE: and HOME:.

class Poly has two additional instance variables, both of them integers. When you create an object of this class, the extra ivars are added to the list of ivars inherited from class Pen. One ivar is for the number of sides of a polygon object created from this class. The other is the length (in pixels) of each side (all sides are of equal length).

The DRAW: method is an extension of the MOVE: and TURN: methods defined in class Pen. First the angle of the turn is calculated by dividing 360 by the number of sides, and is saved in the local variable turnAngle. DRAW: then sets up a DO...LOOP that performs the actual polygon drawing. Using the Sides ivar as the limit for the loop, one side is drawn (GET: Length MOVE: Self). Then the direction is changed by the amount of turnAngle. This draw...turn action is repeated until the index equals the limit of the loop.

SIZE: is redefined for this subclass. It takes two parameters: the length of each side and the number of sides for the polygon. GET: self copies the current pen state into an object's PenState ivars when you specify the size of a new Poly object (SIZE: will be the first selector you'll send to a new poly object, and it needs the PenState variables in its ivars right away). The size parameters are stored in their respective instance variables, Sides and Length. This method also positions an object to the home position (as defined by the HOME: method in class Pen) and orients it facing to the top of the screen (from the UP: method also in class Pen).

The SPIN: method is a routine that draws a sequence of polygons around a center point to make them look as if they are spinning. Notice that this method has one local variable, reps, which is used as a counter for the number of repetitions through the BEGIN...WHILE...REPEAT loop.

Finally, the default settings for an object of class Poly are set by CLASSINIT:. Unless otherwise specified, a Poly object will be a polygon with 3 sides, each 30 pixels long. This method also sets the ivar, maxReps, to 100.

Lines 131 - 136

Next come two examples of objects created from the classes just defined. The first, Bic, is an object of class Pen. Anna is an object of class Poly. In line 136, Anna is changed from its default 30-pixel triangle to a square (4 sides) of 60 pixels on a side.

Experimenting with Turtle

Now that you have an understanding of the inner workings of the Turtle program, it's time to play around with it. We'll start you off with some ideas of things you can do by creating some objects, defining new Mops words, defining new subclasses and even modifying the existing methods to do some tricks. The more you play with Mops, the quicker you will become comfortable with all its powers.

First, you must load the file Turtle. Load it by either selecting the Load command from the File menu, or typing the "slash-slash" command, as in

```
// turtle
```

Now when the file loads, you see a series of dots and occasional messages when words are redefined or if an object name is being reused (is not unique).

Once the files are loaded, you might want to see what Lissajous figures are. Use the Bic pen object as your drawing device. If you look closely at the methods definition for LJ:, you'll see that it needs values in several ivars of Bic for it to function: initLen, deltaLen, deltaDeg, and homeLoc (it also needs maxReps, but that value is set at 200 by CLASSINIT:). For convenience sake, define a Mops word, "lj," that a) takes three input parameters and assigns them to the first three ivars (an operation that is performed by method PUTRANGE:), b) puts the homeLocation in the center of the screen (performed by method CENTER:), and c) draws the Lissajous figures (method LJ:). Here's one way to do it:

```
: lj  cls  putrange: bic
      250 160 center: bic
      lj: bic   cr   ;
```

Try typing in various three integer combinations (e.g., 9 11 301 lj) and watch the variety of curves that are drawn. Try 2 2 2 lj, and you'll notice that the cursor prints on the screen at the last instant before the cr brings the prompt over to the left margin. To eliminate this, you need to turn off the cursor with the Mops word -curs (the opposite, +curs, turns the cursor back on).

Now, define a new word that turns the cursor off before doing the Lissajous figures, and turns it on when the drawing is completed:

```
: cleanlj  -curs  lj  +curs   ;
```

On some integer combinations, the number of repetitions may not be sufficient for the Lissajous figures to complete their drawing (or before they start retracing previous steps). For example, try 12 1 1949. To increase the number of repetitions, you can send a PUTMAX: message to Bic to change its maxReps ivar.

```
1000 putmax: bic
```

Now let's experiment with the Anna object. Right now, it is a square of 60 pixels on a side. Put the coordinates for the center of the screen in Anna's homeLoc ivar by sending a message with the CENTER: selector:

```
250 160 center: Anna
```

Now, move Anna's PnLoc to the center with this message:

```
home: Anna
```

Draw Anna. The square appears on the screen. Now clear the screen (CLS) and resize Anna so the object has 8 sides, each 20 pixels long and draw the object:

```
20 8 size: Anna
draw: Anna
```

In both drawings, the presence of the cursor and Mops prompt really messed things up. Therefore, define a Mops word that: a) clears the screen, b) turns the cursor off, c) draws Anna, d) brings the prompt to the left margin, and e) turns the cursor back on:

```
: draw   cls  -curs  draw: anna  cr  +curs   ;
```

End of lesson 16

Lesson 17

Create a mini-Logo language

The framework established by classes Pen and Poly allow you to create a miniature version of the Logo language, which controls the position and painted trail on the screen of a triangular object called a turtle—hence the name for this demo: Turtle.

We'll show you a few ways to get started. From there, you should be able to develop a rather sophisticated Logo-like environment.

For this experimentation, we will be writing a customized version of Turtle, which we'll call Logo. We'll be using an editor to modify Turtle and save it as Logo for later loading into Mops.dic.

If you have come to this lesson without turning off your Mac or quitting Mops.dic from the last lesson, then you should remove all of Turtle's code from Mops.dic. The fastest way to do this is to use Mops's FORGET operation. FORGET deletes from the current dictionary in memory all the definitions from a word you specify. In other words, you type FORGET plus the first definition of the Turtle program (Pen) to remove Turtle from memory.

To prove it, type:

```
forget pen
```

and then select Words from the List menu. After several lines have printed on the screen, press any key (other than the space key) two times. Notice that the word on the top of the dictionary (the one at the upper left of the listing) is Angle, which is the last definition of Sin—the program loaded prior to Turtle.

If, on the other hand, you are starting this lesson fresh, then start up Mops.dic and load Sin. Our Logo program will load atop Sin.

Designing the language

We start by defining in our minds what we want our mini language to do. First of all, we want a turtle on the screen that will be a triangular object from class Poly. Next, we want to be able to perform a few maneuvers, such as: centering the turtle on the screen; making it move forward in a given direction according to the number of pixels we specify, while the turtle leaves a trail of its pen on the screen; making it turn to the right or left according to the number of degrees we specify. Finally, we'll define one shape, a square, which the turtle will draw if we tell it how long its sides should be.

Looking through the methods available in Poly and Pen, we see that if we draw the turtle in one location and then move it to another, the original turtle on the screen will still be there, cluttering up the screen. Therefore, we need to define an additional method, called UNDRAW:, for class Pen that undraws a turtle where we tell it.

Since the UNDRAW: method will be adjusting the PenPattern (from black to white) and redrawing the object, this method will be defined in terms of the DRAW: method. Therefore, we can place the UNDRAW: method anywhere in the class Poly definition after the DRAW: method.

As far as the PenPattern parameters go, you can look in the QuickDraw chapter of Inside Macintosh for guidance. If there is not enough information there to help (and sometimes there is not), you always have the powers of Mops to help you. For example, while you are experimenting with parameters, you can place a special method inside class Pen that fetches the current values of the parameters from an object:

```
:m INSPECT: \ (  --  HiPat  LoPat  mode w h x y )
      get: PnPatHi  get: PenPatLo  get: PnMode
      get: PnSize  get: PnLoc   ;m
```

Send a message like:

```
inspect: Bic
```

Then perform a .S operation to view the parameters on the stack. Experiment by placing other values in the parameters via a message that calls the INIT: method. Try to draw some objects to learn the results of the new parameters.

Back to the Logo example and UNDRAW:, the PenPattern values that make a white pen are 0,0 while the values for a black pen are -1,-1. Place one integer of the pair in each variable, PnPatHi and PnPatLo, draw the object with a white pen, and then restore the pen to black. The UNDRAW: method could look like this:

```
:m UNDRAW:  \ Erases object before moving it and restore black pen
      0 0  put: PnPatHi  put: PnPatLo  draw: self
      -1 -1  put: PnPatHi  put: PnPatLo  ;m
```

Implementing a Logo-like language

Here is the listing of Mops definitions added to the end of the modified Turtle listing:

```
\ Create Logo-like environment

poly turtle                   \ the name of our Logo object
250 160 center: turtle        \ define the center of the screen
10 3 size: turtle             \ set turtle's size


: SPOT      \ Erases old Logo command onscreen and repositions prompt
      8 210 gotoxy  ;

: .OK  -curs  spot  15 spaces  spot  +curs  ;

\ Shortcut definition for later:

: TURN  -curs  undraw: turtle  turn: turtle  draw: turtle  .ok  ;

\ Logo-like commands:

: HOME    -curs   cls   home: turtle   up: turtle
         draw: turtle  .ok  ;

: FORWARD   \ ( dist -- )
      -curs   undraw: turtle   move: turtle
           draw: turtle   .ok  ;

: LEFT  ( deg -- )            turn  ;

: RIGHT  ( deg -- )      negate  turn  ;

: SQUARE   { len -- }
          -curs   4 0  DO  len forward  90 right   LOOP
      .ok  ;
```

The above Mops words should be self explanatory, except perhaps for the two that control the location of the Logo prompt. In Logo, the traditional prompt location is near the lower left corner of the screen. The Mops word .OK always moves the cursor to the prompt location after the object makes its mark on the screen. The 15 SPACES operation is added to overprint the old command for a cleaner look on the screen.

While in your editor, save the modified source as "Logo" (perform a Save As... operation from the File menu). Return to the Mops.dic window. Select "Echo during load" from the Mops menu. Then load Logo into memory with the Load selection from the File menu or by typing:

```
// logo
```

The program source code will appear on the screen, line by line, as it is being compiled into memory. If you used a word not previously defined, the load will stop, and a message will tell you what word you need to define. Some other messages such as "object not unique" may appear as well. As long as the load doesn't stop, however, nothing fatal is occurring in memory. When the load is complete, clear the screen with CLS and check your program.

Starting the turtle in the home location, try issuing some Logo commands to make the turtle draw lines, turn, and draw squares of various sizes. You'll notice that after turning the turtle to some degree measures (especially those not multiples of 45), the turtle will not fully erase when you issue the subsequent command. The reason is that when the Toolbox draws the turtle at odd angles, the finishing point of the pen may be a pixel off from the original starting point. Then, when the UNDRAW: method is invoked, it undraws from the finishing point of the last operation—off-register from the original motion by one pixel.

But with Mops, that should present no difficulty. Tackle this problem yourself. Try adding another ivar to the object that remembers the starting point of the turtle, and use that point for the UNDRAW: operation. Then, define new Mops-Logo words that make entry of commands easier (e.g., establish abbreviated Logo commands such as FD for Forward). This is the playground on which to cut your teeth on the words in the Mops glossary and class-object-message relationships.

In the remaining lessons of this tutorial, we'll be exploring some of Mops's predefined classes more closely, with the help of an extension of the Turtle program that adds Macintosh-like features to it, such as scroll bars, mouse input, windows, and menus.

End of lesson 17

Lesson 18

Inside grDemo

Before we begin to explain the inner workings of the graphics demonstration program (grDemo), you should be familiar with its basic operation. First, print out the listing for the "grDemo" source file, which is located in the Demo folder. You will need to follow along with the source code listing to understand the discussions in this lesson. It will also be helpful if you have handy a printout of the following files: windowMod.txt, window+, ctl, view and QD.

Next, load and run grDemo to gain an understanding about what it does.

To load grDemo into Mops.dic, double-click Mops.dic to load Mops.dic into memory. Then select Load...from the File menu. When the dialog box appears, select grDemo and open it. This file contains Need commands for the files it needs, and these will be loaded automatically. As each file loads, you'll see a message telling you what is being loaded. When all the files are loaded (the File title on the menubar reverts to black on white, and the flashing cursor appears at the start of a new line), type

```
go
```

Now the program (which we'll call "Curves") will begin. Experiment by moving the scroll bars and selecting different routines from the Graphics menu.

As we explain various parts of this program in these final lessons, we will be revealing many of the Macintosh Toolbox features. While this will in no way serve as a substitute for Inside Macintosh, it will nonetheless give you an appreciation for the many options available to you in programming with Mops. You should also see enough here to guide you to designing other applications.

It is important that you understand the desired results of this program before we explain it, just as it is imperative to know what you want a program to do before you begin writing it. This demonstration program contains some simple examples of menus, controls (like scroll bars) and a window with a separate area in which various graphics routines are displayed. The graphics displayed in the window are the ones defined in the Turtle demo, explained in the last couple of lessons.

If you were designing this program, this would be the time when you ask yourself two basic questions:

(1) What kind of objects will be in the program? What classes need to be defined?

(2) What will be displayed on the screen?

The first question relates more to the inner workings of the program, and the second to the "human interface" aspects. Of course these two questions are interrelated, but it is often helpful at the start to look at them separately.

In the case of our demo program, the question of the inner working is already largely answered, since we will be using the routines we have developed in the last few lessons. The interface question remains, however, and a lot of work needs to be done here. You'll find, however, that Mops has already taken care of most of the messy details.

Views

When thinking about the appearance of anything on the screen, the Mops class which does most of the hard work is class View. In essence, a View is a rectangular area within a window, which displays something. A window is not itself a view, but a window contains one special view which covers the whole of its area except its title bar. This view is called the ContView of the window (since it covers the whole of the window's contents). Everything else which displays in a window must display in a view.

Class View has a number of features which make it quite easy to set the size and position of views, and control what happens when they are clicked or need to be drawn.

Views are hierarchical. That is, views can (and usually do) belong inside other views. We call a view containing another view a parent view, and the view it contains its child view. If you have done any Newton programming, you'll be familiar with this terminology. In fact, you'll be pleased to know that many of the Mops view features will look rather familiar.

Let's look again at the Curves window to identify the views.

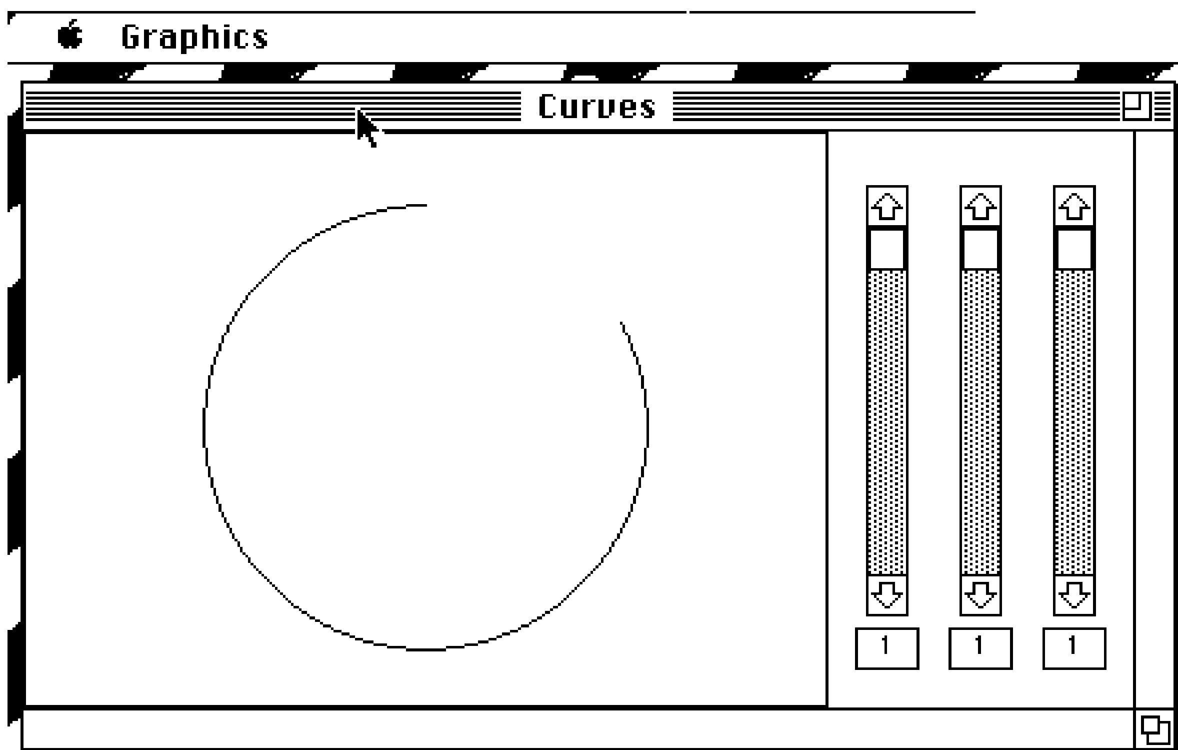


Figure I-17

This window has a pane in which the graphics are drawn—clearly this will be a view. There are also the three scroll bars and the digital displays beneath each one. The scroll bars will be views, and the digital displays will also be views.

Positioning views

So far so good. Now we also need to look at how these views will be positioned. Clearly the digital display which belongs to each of the scroll bars should stay with its scroll bar, even if we reorganize things so the scroll bars go somewhere else in the window. The simplest way to do this is to create some more views. If both a scroll bar and its digital display are child views of one parent, it becomes easy to move them together. So what we do in the Demo program, is to define three "indicator" views, where each indicator has a scroll bar and a

digital display as its children.

There is one more view, and this is the contView for the window—the view which covers the whole of the window's area.

Thus we have 11 views in all. The contView is at the top level. Then the graphic pane and the three indicators are at the next level—these are the children of the contView. Then each of the three indicators has two children —a scroll bar and a digital display.

The position and size of a view is defined in terms of its bounding rectangle, its viewRect. However in a program you won't normally set the viewRect directly (although you can if you have to). Normally you will specify the four sides of this rectangle relative to the view's parent or siblings, with a range of options, which can be independently set for the four sides.

The measurement option for each side is called its "justification", and the corresponding number the "bound". There are four justification values and four bounds for each view. If a parent view moves or is resized, normally its child views are also moved/resized according to their bounds and justifications.

Again, this scheme will be familiar to Newton programmers. So will most of the justification options we provide, although we do have a few extras. One such extra is that we have four justification values for each view, one for each side, instead of just two (for horizontal and vertical)—this reflects the fact that views are more likely to be resized while a program is running, on a Mac than on a Newton.

Here are the possible justification values (defined as constants):

Name	**Meaning**	**Description**
parLeft	parent left	the bound is measured from the left edge of the parent.
parRight	parent right	the bound is measured from the right edge of the parent.
parCenter	parent center	the bound is measured from the center of the parent.
parProp	parent proportional	the "bound" is a value out of 10000, expressing a proportional distance across the width of the parent.
parTop	parent top	the bound is measured from the top of the parent.
parBottom	parent bottom	the bound is measured from the bottom of the parent. (Note that parCenter and parProp can also be used in the vertical direction, with the obvious meanings)
sibLeft	sibling left	the bound is measured from the left of the previous sibling.
sibRight	sibling right	the bound is measured from the right of the previous sibling.
sibTop	sibling top	the bound is measured from the top of the previous sibling.
sibBottom	sibling bottom	the bound is measured from the bottom of the previous sibling.

The default values are parLeft and parTop, which simply means that the child view's bounds are relative to the top left corner of its parent.

This scheme may look complicated, but is really quite easy to use. In most situations the bounds and justification values can be set up for a view at compile time (via the setJust: and setBounds: messages), and your program won't need to take any other action—the view will keep moving itself automatically to the right position whenever the parent view moves.

Let's look at how we've set up these quantities for the views in our grDemo application. Look at the CLASSINIT: method in class Indicator:

```
:m CLASSINIT:
      parCenter   parTop      parCenter   parBottom   setJust: theVscroll
      -8          0           8           -20         setBounds: theVscroll
      parCenter   parBottom   parCenter   parBottom   setJust: theReadout
      -12         -16         12          0           setBounds: theReadout
      classinit: super
;m
```

Here we position the child views of the Indicator, namely the vertical scroll bar theVscroll, and the box with the digital display, theReadout.

Note that both setJust: and setBounds: methods take four parameters, which are in the usual order left, top, right, bottom.

In the horizontal direction, we want both the child views centered within the Indicator (theReadout will be directly underneath theVscroll). The width of a scroll bar is always 16 pixels, so we set the left and right justification to parCenter, and its horizontal bounds to (-8, 8). Our readout box is going to be 24 pixels wide, so again we set the left and right justification to parCenter, this time with the horizontal bounds (-12, 12).

In the vertical direction, we want the top of the scroll bar to coincide with the top of the Indicator, and the bottom to be a fixed 20 pixels above the bottom of the Indicator, to allow room for the readout box. So we set the top justification to parTop and the top bound to 0, and the bottom justification to parBottom with the bound -20. The readout box will have a fixed height of 16 pixels and always be right at the bottom of the Indicator, so we set both top and bottom justification to parBottom, with the bounds (-16, 0).

If you try resizing the demo window, you'll see the results of these settings. The vertical scroll bars will stretch or shrink according to the height of the window, while the readout boxes will remain at the same size.

We should mention why we put the setting of these values into the CLASSINIT: method of Indicator. It is primarily because theVscroll and theReadout are ivars of Indicator. This strategy makes it easy for us to have more than one Indicator. At the moment we have three, but it would be very easy to have more.

In the case of the graphic display view, dPane, the three indicators themselves, and the contView of the window, dView, we simply declare them in the dictionary. We certainly could have made them ivars of the grWind class, since they really belong to the grWind object. This would have been advantageous if we had a number of grWind objects instead of just one. However if we had done things this way, the grWind class would need methods so that the views could be accessed from outside the grWind class. In our example here, since we have just one grWind object, it is a bit easier to declare the views as normal objects in the dictionary.

This is an example of the kind of tradeoff that will frequently occur in designing a program. In Mops, however, you will find that it is not very difficult to change your design in various ways even after much of the program is written. This is one of the benefits of object-oriented programming.

Since we declared these views in the dictionary, we don't need to set them up in grWind's CLASSINIT: method, and can simply set them up directly at compile time. If you look at the lines where we do this, you should be able to work out how we these views are positioned. We also give some alternative code, commented out. If you use these lines instead, the Indicators will be evenly spaced along the bottom of the window, instead of at the right hand side. Note how we use the parProp justification to achieve the even spacing.

Drawing views—the DRAW: method

When a view is to be drawn on the screen, it will receive a DRAW: message. The default behavior is to execute the draw handler, which is an X-addr (execution address), which can be set to execute any word. There are a number of other housekeeping actions to be done as well, which Mops does automatically. When any view has its DRAW: method called:

(1) The screen coordinates will have been set so that the top left corner of the view is at (0,0). In Mac terminology, this is called setting the origin to be relative to the view.

(2) The clip region will have been set to coincide with the view. Thus you can draw outside the view, but anything outside will not appear on the screen. This can simplify things considerably, since you can draw in a view without having to worry about its exact size at that time (and of course its size might change while the program is running).

(3) Mops has a Rect object, tempRect, which is used for a number of things. When DRAW: is called on a view, tempRect will have been set to coincide with the boundary of the view. This can be very handy. We use this feature in the demo to draw a frame around the graphic, and to draw the readout boxes.

The other thing that DRAW: does is to DRAW: on all the child views. This way, when a window has to be drawn, all we have to do is ensure that the contView gets a DRAW: message. This will ensure that all views get a DRAW:, and if everything has been properly set up, the window will be fully drawn.

Your code to draw a view can be placed in the draw handler, or you can subclass View and put the code directly in the DRAW: method of the subclass. This is probably easier, in general, and is what we do in the Readout views.

Look at the DRAW: method for class Readout now. Note how we exploit the fact that tempRect is set to the view's boundary, in local coordinates (which are in effect at this point). Here we erase the previous number that was being displayed and draw the box around the view. Next, the "cursor" where the digits are to be placed is positioned three pixels across and 10 pixels down from the top left corner of the rectangle.

Now the new digits are printed in a field of 3 digits. First we set the set the textmode to 1, the textsize to 9, and the textfont to number 1. Textmode determines how the pen that draws the numbers on the screen will react to the color of the screen below it. With the mode set to 1, the pen draws black on the white background. The textsize number is the actual font size, like the sizes you select in the MacWrite Font menu. The textsize setting of 9 calls for 9-point type.

The textfont number requires a little more explanation. In the Mac Toolbox, fonts are assigned ID numbers. Some commonly used ones are as follows:

SystemFont (Chicago)	=	0
ApplicationFont (Geneva)	=	1
New York	=	2
Geneva	=	3
Monaco	=	4
Times	=	20

While in this list the application font is the same as Geneva, which is the default, in some programs, a special application font is inserted in its place. Since we have not done this, our application font will be Geneva. For the digits in the readout box in grDemo, then, the Geneva font was selected.

Now look at the DRAW: method for class Indicator. It gets the current value from the vertical scroll bar, then sends that to the Readout view via Readout's PUT: method. Finally it calls (DRAW): super which does whatever its superclass needs to do for drawing. As we'll see later, we don't use DRAW: super here, because of the other automatic actions such as setting the clip which are done by DRAW:. These actions have already been done for this view, and don't need to be done again. And note that one of the things that we don't need to do again is to call DRAW: on the Readout view, since the Readout is a child view, and it will get a DRAW: automatically.

The three scroll bars are objects of class VScroll, which is a subclass of Control, which is a subclass of View. Mac controls are drawn by the system. We handle this simply by defining DRAW: for class Control to make the appropriate system call. You won't normally need to make any changes to this behavior.

The NEW: method in class Indicator is called at run time, when the window containing the view is opened. Here we need to set up which are the child views of this view. This is done with the ADDVIEW: method. We can't do this at compile time, since we have to pass in the address of the child view we're adding, and this might be different in different Mops runs.

Finally, notice that the graphics drawing view, dPane, is an instance of a one-off View subclass, Pane. For Pane, we don't write a DRAW: method. This is because DRAW: for the superclass, View, already does exactly what we want. For dPane, we need to change the drawing behavior while the program is running, as the user selects different options from the menu. So in this case it is better to set a draw handler when a menu choice is made, via the SETDRAW: method (This is a good example of a situation where the use of a draw handler is appropriate.) The draw handler is called as part of the action of DRAW: for class View, so we don't need to modify this at all in class Pane.

In fact, Pane only has one method overriding what the superclass View does, and that is the MOVED: method. MOVED: is called when the bounds or justification has changed, or when the parent view has moved, as any of these occurrences will require the view's position and size to be recalculated. MOVED: in class View does this recalculation and resets the view's viewRect appropriately.

Often when a view moves it will need to erase its old position, and this is the case with our drawing pane. Therefore our MOVED: method just clears itself (it is still at the old position at this time), then calls moved: super to do the actual moving.

End of lesson 18

Lesson 19

Windows

Let's now return to the listing for grDemo. After we set the bounds and justifications for the dPane and the three Indicators, we declare dView, which is to be the contView of our window. This can be a plain View, since it doesn't need to have any other special properties.

Five values are created next. The first pair are the coordinate point on the Mac screen of the top left corner of the window that the program will occupy: coordinates 40,60. The next two are the coordinates for the right bottom corner of the program window. These values will be suitable for the small Mac screen (on e.g. a Classic), but should look all right on any Mac screen. These figures will be recalled later when it comes time to create the window for the program.

Next we come to a class that defines a special kind of window: one that has controls in it and has an area where graphics will be drawn.

The Macintosh Toolbox contains six predefined windows, each with a unique window definition ID number. The six windows, their names, and their IDs are illustrated in Figure I-19.

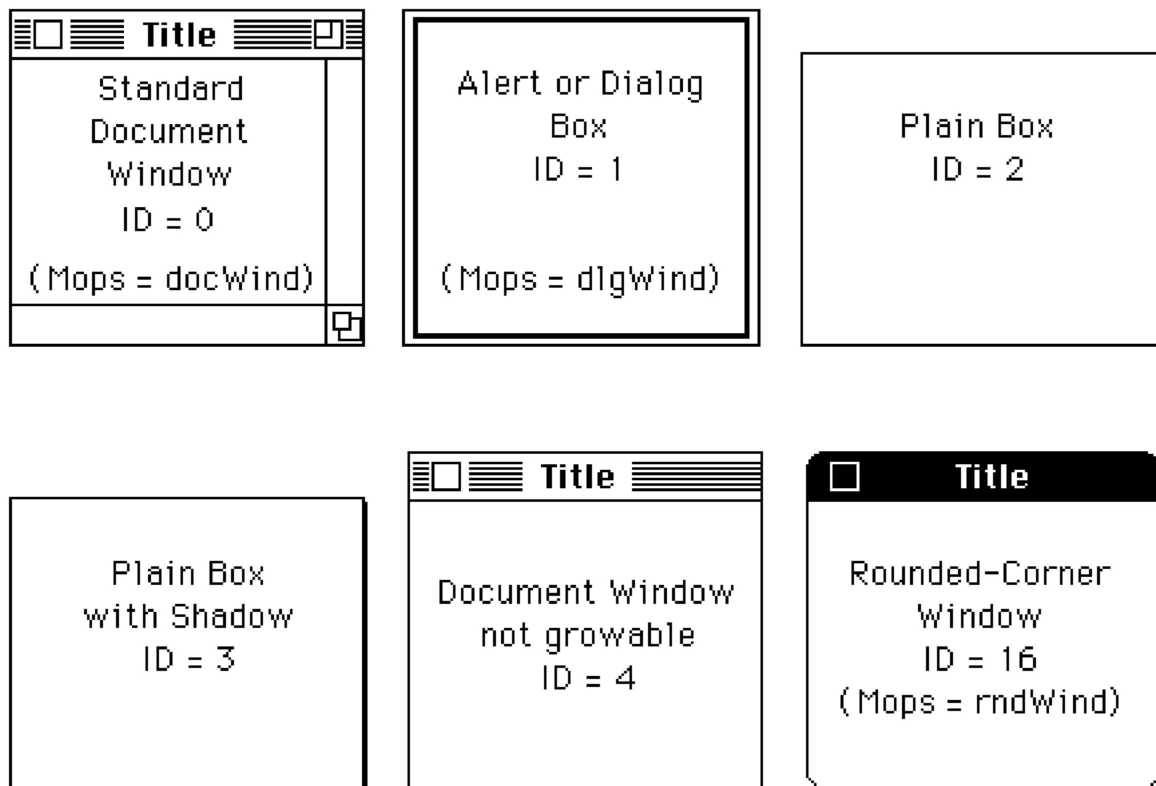


Figure I-18

Whenever you define a new window, choose one of the window types by number. Mops has established three constants—docWind, dlgWind, and rndWind—that you can substitute in place of the number, in case it's easier for you to remember names than numbers. The Mops constant names are shown in Figure I-19. This Mops demo graphics program uses docWind, since Apple's Human Interface Guidelines recommend we use this for a standard growable window.

Even plain windows are relatively complex objects inside the Macintosh Toolbox. To give you an idea of their complexity, look at the long list of instance variables in the predefined class, Window (in the file WindowMod.txt, in the folder "Module source"). Among the items you can control—and sometimes must control—in a window are:
• the kind of window (from an inventory of six)
• the rectangular area on the screen to enclose the window
• whether a window is growable
• the area on the screen within which a window can grow
• whether a window is draggable
• the area on the screen within which a window can be dragged
• how it is to respond to key-down and mouse-down events

The grWind class

class grWind, a subclass of Window+ (itself a subclass of Window) lays the framework for the window of this graphics demonstration. As we have seen in the last lesson, all Window+ objects contain a view, the contView, which simply covers the whole area of the window. The window knows which view is its contView, since this is specified when the NEW: message is sent to the window object to fire it up. In our example, the contView of our grWind window is called dView.

Giving the window enough parameters to present itself on the screen is simplified in this program in the NEW: method, which is an extension of the NEW: method of class Window+. The only parameters needed are the address and length of the title of the window. The other pieces—the address of the window's rectangular bounds, the type of window (rndWind), flags for being visible on the screen and having no close box, and the address of the contView—are supplied within the method or as constants already defined. Once all the items are on the stack in the proper order, the method calls the superclass' NEW: method.

One of the actions performed by the NEW: method in class Window+ is to send NEW: to the contView. This will cause NEW: to also be sent to every view in the window, since one of the actions of NEW: in class View is to send NEW: to all its child views.

Once the window and all its views have been fired up, the views will draw themselves. As we have seen, views draw themselves when they get a DRAW: message. And what do we have to do to start this process off? You may remember we mentioned this in the last lesson, but by now you might be able to guess it anyway. The answer: we have to send DRAW: to the contView!

On the Mac, drawing in a window normally takes place when a window is updated. Drag the grDemo window to the bottom of the screen so part of it runs off the screen. Release the mouse button. Now drag it back near the center of the screen. For everything in the window to be visible again requires updating. When the Mac system recognizes that a window has to be updated, it sends an "update event" to the application which owns the window. You don't need to worry about the details at this stage, but Mops catches this update event and sends a DRAW: message to the window.

And this is how the contView gets the DRAW: message that will cause all the views to be drawn. The DRAW: method in the Window+ class basically just sends DRAW: to the contView. Easy!

dWind

Continuing with the grDemo listing, straight after our definition of the grWind class, we define our window, dWind, which is an object of the class grWind we just defined.

A new definition, @DPARMS, gets things ready for the drawing of the graphics in the view dPane. This definition is a shortcut that allows us to use one word to do work which is needed for each of the four following definitions.

The first thing we have to do is erase whatever was in the view area before, and then we have to draw a rectangle as a border around the view, Now remember that the graphic drawing will be done as a result of a DRAW: message being sent to dPane, and that at this time the rectangle tempRect will have been set to the border of the view. This makes our job easy—we simply write clear: tempRect draw: tempRect.

The other task for @DPARMS is to fetch the current readings of each control.

The words of the four following definitions should look familiar. The definitions are extensions of the spiral, spin, lj, and dragon curves defined in Turtle. Here, however, they have been modified to fetch three control parameters, place those numbers as ivars of the drawing device (the pen or poly, as the case may be), and draw the graphics accordingly.

You will notice that instead of using : and ; in these definitions, we have used :a and ;a. We did this because these words are action handlers—their addresses will later be stored in a Menu object, to define the actions that are to take place when the user makes a menu selection. We will see in a later section when discussing modules, that action handlers in modules need to be declared with :a and ;a. This is not strictly necessary in the main dictionary, where we are now, but it does no harm either. It is therefore a good idea to always use :a and ;a for action handlers, as it will make your programs clearer and also make it easier for you if you later move code into modules.

Because each of the drawing types has a different range of parameters, the !ranges definition lets us set the maximum number for each control, depending on which graphics type we select from the menu. The minimum values are always one. (We will discuss controls in more detail shortly.)

In the next three lines of code, the text of the message that appears on the screen in response to the "About Curves" menu items is assigned to two string constants, AB1 and AB2. Following that come three program lines that define what is to happen when that selection is made. It selects the font Times in 14-point (Times has font number 20), positions the cursor at point 28,40, and "types" the two strings on the screen. It then calls the word WaitClick, which will just wait for the user to click the mouse or type a key. Then it sends an UPDATE: message to dWind to cause it to redraw itself, erasing the text we "typed".

Next, both the pen Bic and the polygon Anna are told where the center point of the graphics rectangle is located. Importantly, the coordinates given are relative to the top left corner of the view dPane, since that is where the origin will be whenever dPane receives a DRAW: message.

Controls

In the Macintosh environment, a control is a screen object that responds to interaction from the mouse in such a way that the mouse causes either instant action or a change in function for a later operation. A good example of the "instant action" kind of control is the elevator knob on the volume control in the Sound Control Panel. By adjusting the knob with the mouse, you immediately adjust the volume of the sound played by the Mac through its speaker. Likewise, when you click an "OK" button in a dialog box, you are working with a control for immediate action. A "delayed action" control would be something like the check box inside a Get Info dialog window that locks or unlocks documents for dragging to the trash. When you click the mouse pointer in an empty box, an "X" fills in the box, and the document is locked, but no particular action occurs in response. Click the pointer again, and the X disappears, so you can go ahead and trash the document.

In Mops, controls are View subclasses. Thus, they are sized and positioned via the justification and bounds quantities we talked about in the last lesson, and they are drawn when they get a DRAW: message. But as well as this, they have a number of interactions with the Mac system.

A scroll bar is one of the most common kind of control. It consists of five parts, each of which responds differently in the course of a program. The five parts are:

```
Up arrow
Page Up region
Thumb
Page Down region
Down arrow
```

Each region is programmed to respond as needed.

Like many objects that the Macintosh Toolbox predefines, controls have specific identification numbers, called control definition IDs, which tell the Mac what function the control is to play and how it is to look. The four standard control types and their definitions IDs are:

simple button	=	0
check box	=	1
radio button	=	2
scroll bar	=	16

All controls also need to specify actions based on their interaction with the mouse. Scroll bars, with their five distinct parts, need separate actions specified for each part. An action is nothing more than a set of instructions to follow when a control part is activated by the mouse. In a Mops program, the actions, or rather the addresses of the action definitions, are stored as instance variables of a control object. Moreover, each control part has a distinct ID number so the Toolbox knows to link a given action with a given mouse interaction. The IDs for all Macintosh predefined controls are as follows:

simple button	=	10
check box or radio button	=	11
scroll bar Up arrow	=	20
scroll bar Down arrow	=	21
scroll bar Page Up region	=	22
scroll bar Page Down region	=	23
scroll bar Thumb	=	129

GrDemo scroll bars

The scroll bars in grDemo inherit their instance variables from the superclasses View, and Control. The list of available ivars includes an integer for the definition ID, an X-array for the addresses of a scroll bar's five possible actions, and an Ordered-col for the actions' corresponding part numbers.

Whenever a VScroll object is created, its CLASSINIT: method automatically makes it a scroll bar by putting the control ID number 16 into its ID ivar. The method also places null values in each of the object's actions. When it receives a NEW: message, which all views do when they are to display themselves, it makes the appropriate call to the system to cause itself to appear. It also makes this call whenever it gets a DRAW: message. (The definition for the VScroll class is in the file Ctl in the Toolbox Classes folder.)

Scroll bar actions

Continuing down the demo program, the list of 5 definitions are the actions that occur when you click each part of each scroll bar. The formats for each action handler definition is much like the other except for the amount of increment. A key element of these definitions, however, is that they call upon a special Mops construction, called MyCtl.

MyCtl is what is known as a vector. MyCtl essentially tracks the address of the most recently activated control. Therefore, if you click the PageUp part of the second of our three scroll bars, MyCtl remembers that it was the second scroll bar you activated. In the action handler definition, then, get: MyCtl fetches the previous value of the second scroll bar. The object of the get: method is determined dynamically at runtime, a technique explained in Part II as late-binding. After the value of the second scroll bar is decremented by 10, a put: MyCtl stores the value in the second scroll bar's ivar before sending the update message to the window. The importance of this myCtl mechanism is that is eliminates the need for us to define five action handlers for each scroll bar or concocting some algorithm to keep all that code to a minimum. MyCtl allows us full control flexibility with a minimum of code.

The doThumb definition is a special one that automatically calculates a value based on the relative position of the thumb along the range of the scroll bar. The doPgUp and doPgDn increment and decrement (respectively) the value of the scroll bar by 10. And the doLnUp and doLnDn adjust the figure by one in their respective directions.

In the line after the action handler definitions, the address of the lj definition (the one that draws Lissajous figures) is plugged into dwind as the type of graphic that gets drawn when grDemo first fires up. The notation ' ("tick") returns an execution token or xt, a quantity which can be saved and used later to execute the word. In this case, the xt of lj, which was defined a bit earlier in this program, is passed as a parameter in the setdraw: dwind message. Checking at dwind's class definition, we find that the setdraw: method stores the xt of a graphics routine (lj, spin, etc.) in the draw ivar of dwind. This will all come together at the end of the program.

A small digression: in Mops, as in many Forths, this execution token is actually the address of the compiled, executable code for the word. However this isn't true of all Forths, and so the new ANSI standard now uses the more general term "execution token".

Next, the xts of the five control actions are stored in each scroll bar's actions ivars. The syntax here, xts{ ... } is a shortcut for entering the xt of each action handler word. Addresses for each definition are passed as parameters to the scroll bars' actions ivars.

End of lesson 19

Lesson 20

Menus

A Mops program uses menus which are stored as separate resources. This is the normal Mac method for defining menus, since it makes it easy for users to customize the menu text with ResEdit.

Menus work in a way analogous to controls in that the program contains definitions of menu handler words, which the menu selections invoke. Menu selections are usually more powerful in a Mac program than controls, because menus typically divert the program into a relatively drastic change in program mode. In a typical File menu, for example, selecting the Load... option halts the main program, while the user's attention is shifted to the dialog box for the selection of a file to open. In grDemo, the primary menu, Graphics, changes the type of graphics the program will draw, sending you from Lissajous mode to Dragon Curves mode, for example.

Menus take a bit more setting up than controls, since there are two separate steps. First, you have to create a resource file to contain the menu resources. This can be done with ResEdit. For the demo program, we have provided a resource file called "Demo.rsrc". This is in the Mops ƒ folder. Then the second step is to set up your Mops menu objects to correspond with the resources—we'll talk about this now.

Menus have an ID number associated with them. It's important to note that this menu ID is different to the resource ID that menu resources have. The existence of two separate ID numbers for a menu frequently causes confusion, but this can be minimized if you normally make these numbers the same.

By convention, the Apple menu is ID=1. We've assigned ID=2 to Graphics. These ID numbers are stored in the menu resources, and are used by the Mac system to identify which menu has been selected. You will see now that in Mops there is a simple way of associating a set of actions with a particular menu, using the menu ID number to identify the menu.

Returning to the grDemo program listing, we now define a menu object called grafMen, specifying that it has six members. AppleMen does not need to be defined here, because Mops has already defined it in Mops.dic.

Next come the menu handler word definitions for grafMen. Each one places the xt of the drawing word in the draw ivar of dwind. Yet another syntax for obtaining the xt of a word is demonstrated here: ['] lj. ['] is the equivalent of ' <u>within a definition</u>. Each definition also places the maximum control values for each type of drawing. Then it sends an update message for the entire window, which draws the revised scroll bar values and the drawing for the current settings.

SetReps is a word that establishes the maximum number of repetitions for drawings created using the pen bic and the polygon anna. You may wish to increase the value for bic if you find your numeric selections on the scroll bars don't draw complete figures. Conversely, some drawings may repeat on themselves after only 100 or fewer repetitions, in which case it seems that the program is unresponsive for several seconds.

Next we set up the menus. We do this by sending an INIT: message to each menu object. As you can see, this method takes a xt list on the stack, which we specify using the xt{ ... } syntax. It also takes another number, which is the menu ID number we've already talked about. The xts in each list refer to the words which will be executed when a selection is made from the menu, starting from the top item. Thus if the first item of GrafMen is selected, the word that will be executed is doLiss. This corresponds to the item "Lissajous" which is in the menu resource. It is your responsibility to make sure that the words that you put in your xt lists correspond, item for item, with the text that you have put in the items in your menu resource. The Mac has no way of knowing that doLiss corresponds to the text "Lissajous". If you get the xts in the wrong order, you will get some interesting things happening when you make menu selections, but it won't be what you want!

In the grafMen resource, we have included a gray line between "Dragon curves" and "Quit", which (as is customary) is the last item. Even this gray line must have a corresponding xt in the list. You can use any word at all here, since it will never be executed; however to make our intention clearer we have used the word Null, which is a word which does nothing anyway.

You will notice that for the Apple menu, appleMen, we have put only two items in the xt list. All right, what about all the dozens of items that may be sitting under your Apple menu? We are actually taking care of them, with the use of another feature of Mops—if there are more items in a menu than were present in the xt list, and one of the "excess" items is selected, the last xt in your list is called. The word DoDsk handles the firing up of a DA or whatever is under the Apple menu. In our program here it will handle everything except the first item, which (as is normal Mac practice) is "About...", in this case "About Curves".

Running the program

The last definition of this program is that of a word that gets the whole program running. This is where everything done so far comes together when you type the word, GO.

In the first line we do a couple of things if the value Instld? is False. Now this value will be True if this is a stand-alone (double-clickable) application, and False otherwise, that is if we have just loaded this program into the Mops dictionary. (We will see in the next lesson how to install our demo program as a double-clickable application.)

These two actions, then, are things we need to do when testing our program in Mops. The first action is to open the resource file, demo.rsrc, which contains the menu resources we need. In a stand-alone application we don't want to carry around a separate resource file, so our normal practice will be first, to create the stand-alone application via Install, and then to add any extra needed resources with ResEdit. In this case the resources will be available without any other file having to be opened. During testing, however, it is more convenient for extra resources to be in a separate file.

In the next line, we bring our window, dWind, to life with a NEW: message. We pass in the address and length of the text that will appear as the title of the window. The syntax " Curves" (with a space after the first ") compiles the text "Curves" into the dictionary, and at run time pushes the address and length of the text on to the stack, which is what we need to pass with the NEW: message.

Next we fire up the Menu objects. The GETNEW: messages sent to each Menu object gets the needed Menu resources and initializes the menu. We then set up the menu bar (at the top of the screen) by sending an INIT: message to our Menubar object. This message takes the menu object addresses on the stack, followed by a count of the number of menus.

Newobjs, as defined earlier, brings the dwind and VS objects to life. Bic and Anna have their respective maximum repetitions set, and the cursor is turned off. At the end, the word EventLoop enters an endless loop

which continually "listens" to mouse events as they affect controls and menus.

Last of all we define the error word. This is needed for an installed application, since the full Mops system won't be there, so you have to provide a word to be executed if some error arises which normally gives a Mops error message. We customarily call this word CRASH, which is a good description! Here it just beeps twice and quits to the Finder. A "real" application would need to do something more helpful, probably with an alert box, but this is, after all, a demo.

If you want the program to start up right away after loading, all you have to do is enter the startup word, GO, as the last word of the grDemo source file. When the file is loaded, Mops will act on that startup word as if you had typed it at the Mops prompt.

In summary

Now that you have seen the entire grDemo program, you should notice some key points about Mops programs. First come the definition of the classes of objects that appear on the screen. The balance of the program concerns itself with defining handler words that work their wonders when controls and menus are activated by the mouse. It is wise to think of your program action in terms of handler words. And lastly comes the definition of the word that starts your program. it calls the words you've defined in the dictionary to create objects and let the program respond to your input.

End of lesson 20

Lesson 21

Installing an application

In this lesson we will install our Curves program as a double-clickable application. First, load GrDemo as we described in lesson 18. But this time, don't start it up. Either type

```
install
```

or select Install... from the Utilities menu. Either way, the following dialog box will appear:

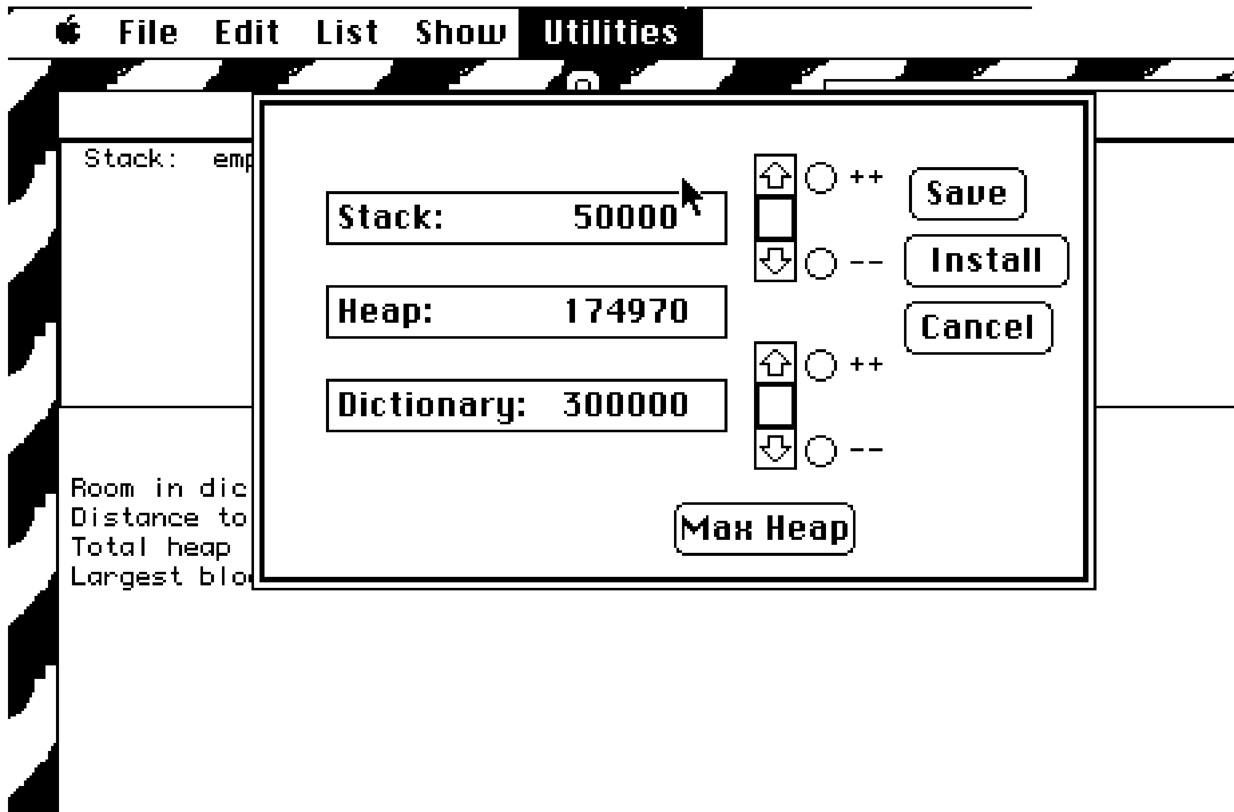


Figure I-19

Experiment with clicking in the controls. (Don't click the buttons yet.) You will see that if the Dictionary number increases, the Heap number reduces by the same amount, and similarly for the Stack. You are here defining how the available memory will be used in your application. The "stack" space is for your parameter stack and your return stack. The system will also use the parameter stack space when various system calls are made, which can use a lot of stack space, so we suggest you don't reduce this figure below 20000.

The Dictionary number refers to the amount of memory allocated for the dictionary, above what is allocated already. What we are doing now is installing an application which is already loaded, so that we won't need a very big number here, since no more definitions will be getting compiled into the dictionary when our application is running.

The Heap number refers to the memory that is available on request from the system, when your program is running. The number here is the maximum amount of this kind of memory that is available. Basically whatever is left over after the stack and dictionary space is allocated will be used for the heap. The number that appears here is only a guide, since when your installed application runs there mightn't be the same total amount of memory available as there was when you ran the install. It is good, then, to use only what you really require for the stack and dictionary, to make the best use of whatever memory is available when your application runs.

Normally the easiest way to do this when you are installing an application, is to click on the "Max Heap" button. Do it now. Your window should now appear something like this, perhaps with a different number for the Heap:

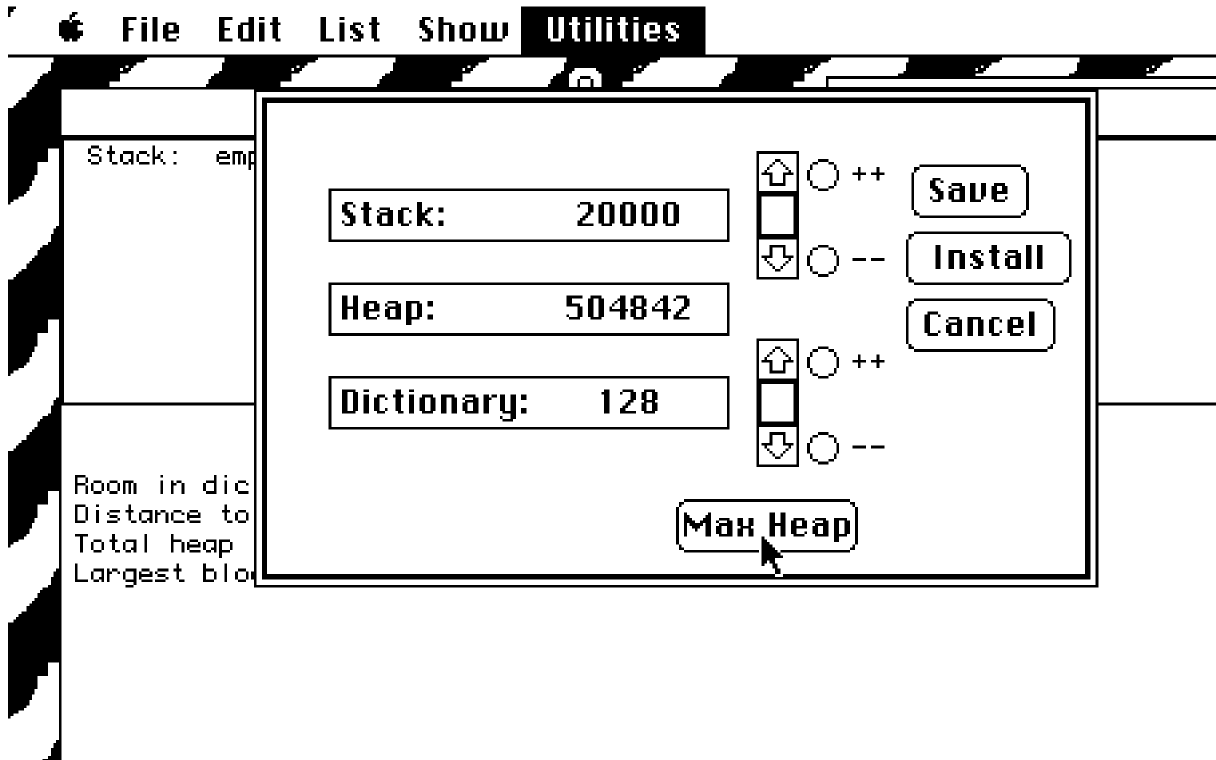


Figure I-20

You will see that the stack allocation is now 20000, which is the mimimum we recommend, and the dictionary only 128. This just allows a safety margin in case your application executes code that moves a string to HERE. (If you know that your application will need more room at HERE, you can adjust this number. For the demo program, however, there's no need.)

Now that you have specified your memory requirements, click Install. You will get the following dialog:

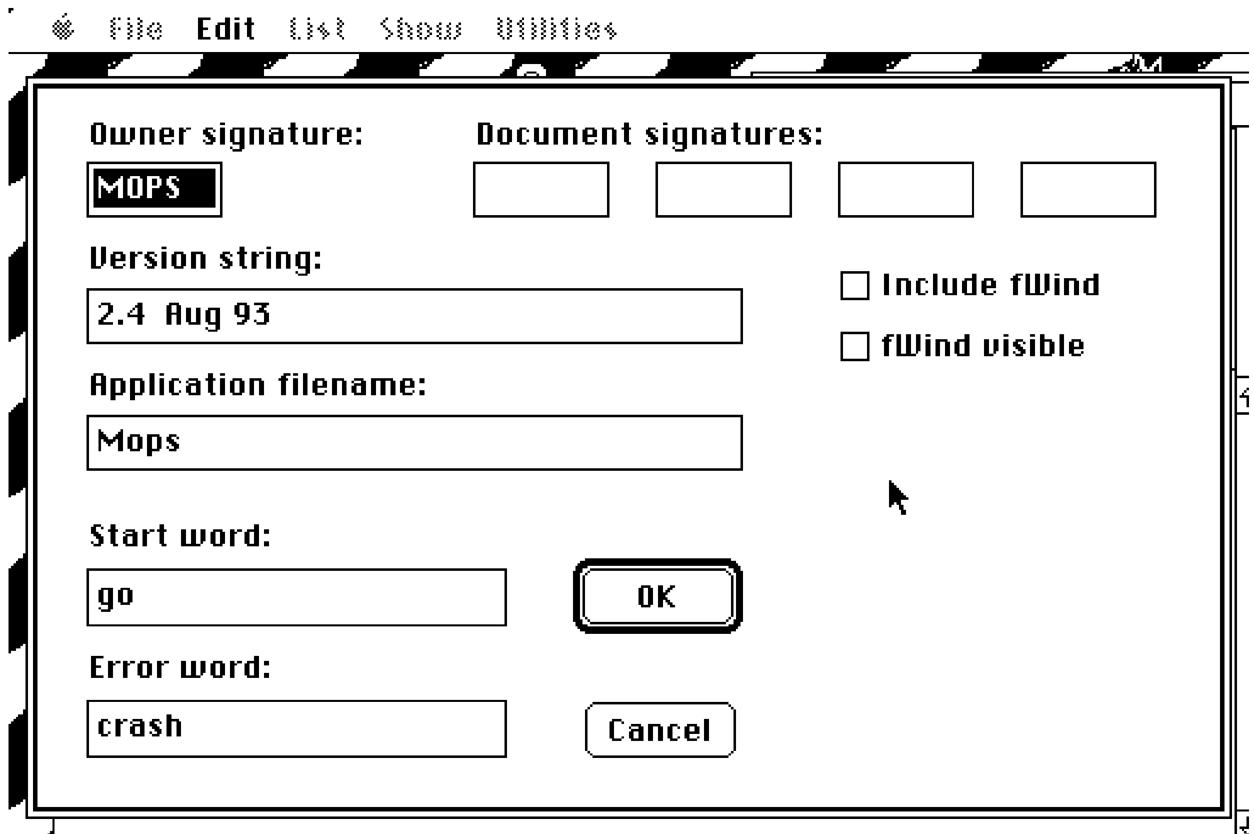


Figure I-21

This dialog allows you to set your application and document signatures. These are both 4-character quantities, and are described in detail in Inside Macintosh. If you are going to release your application widely, you will need to register these signatures with Apple (this avoids having different programs using the same signatures). For your private use, however, you can use any four upper-case letters you like. For this demo program, Curves, we've chosen CRVS. Curves will have no documents of its own, so leave the other boxes blank. Type the name of the application, Curves, in the appropriate box, and anything you like for a "version string". In the example here, we've put "version 1" which seems to make sense. The dialog box will now look like this:

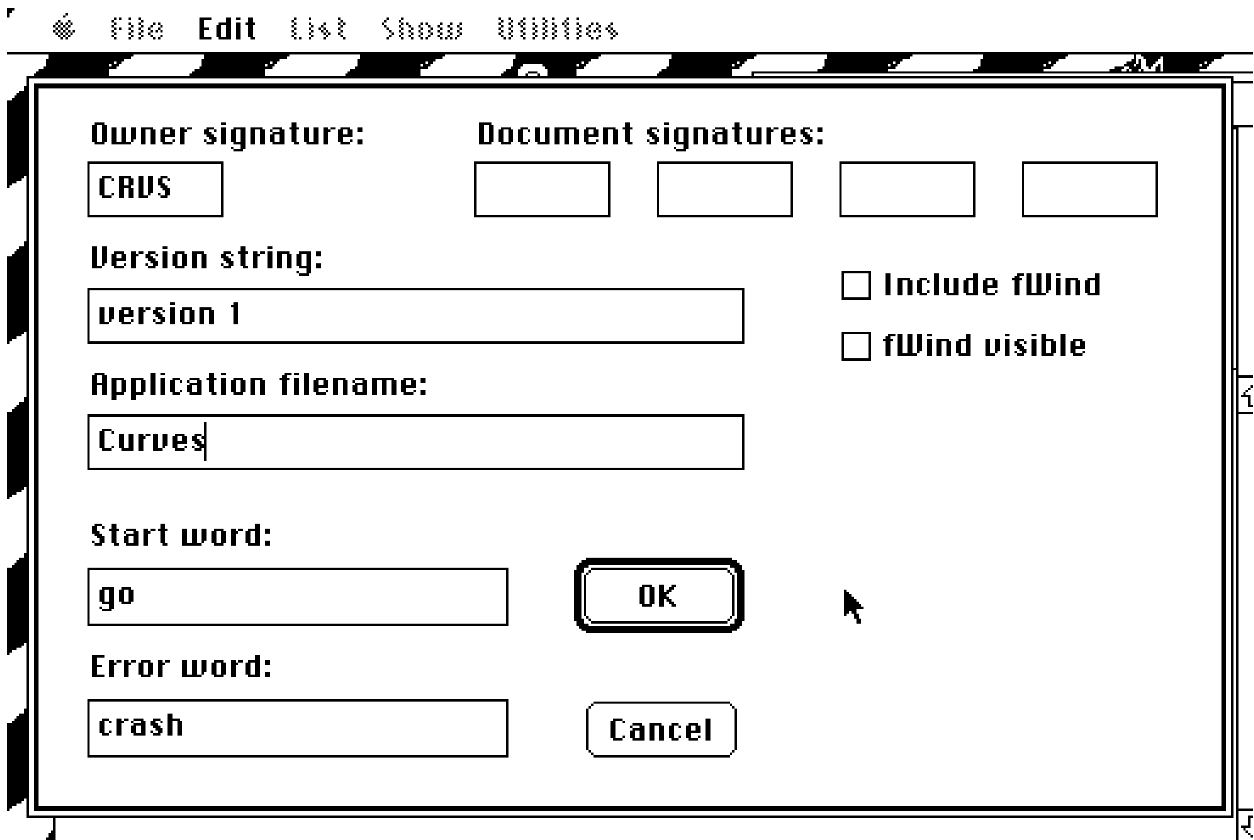


Figure I-22

We explained the "start word" and the "error word" in the last lesson. The dialog suggests the names GO and CRASH respectively, which are, in fact, the names of the words we've used, so we can leave them unchanged.

The way these words are handled in an installed application is quite simple, thanks to the mechanism of vectors, which we introduced in lesson 20. A vector is basically the same as what some Forth systems call a DEFERred word. A vector contains an address. You call a vector in the same way as for an ordinary word, but it is the word where the address points which is actually executed. The address can be changed at any time. The start word and the error word addresses are put into two vectors, QuitVec and AbortVec.

QuitVec is executed whenever the word QUIT is executed, which is at the start of each time around the Mops interpretation loop (the loop which waits for keyboard input, then executes it). Normally QuitVec does nothing (it points to NULL), but in an installed application it is set to point to the start word. The start word should loop indefinitely, handling incoming clicks or whatever, and never terminate itself. Of course the application will eventually terminate, but this should be in response to some user action which is being handled by some word called from the start word—what we mean is that the start word should not exit through its ending semicolon. If it did, the rest of QUIT would be executed, which would attempt to interpret keyboard input as Mops words. This is definitely not what you want in an installed application—for one thing, the Mops window would probably not be there.

AbortVec is called when Mops detects an error which normally gives a Mops error message. Like QuitVec it initially points to NULL. In an installed application you don't want your users to see a stack dump (and anyway the Mops window probably wouldn't be there at all), so your error word should do whatever is appropriate for your application, and perhaps then execute BYE to quit to the Finder. Like the start word, it shouldn't exit through its final semicolon, since that would lead to Mops trying to give a Mops-style error message and stack dump.

Now to continue with the dialog box. Leave "Include fWind" and "fWind visible" unchecked. These refer to a simple window for keyboard input and text output which can be used for "quick and dirty" applications. (This is actually the window used by the basic nucleus before the rest of the Mops system is loaded.) Curves makes no use of this window, since it has its own, so by leaving these boxes unchecked, we are telling Install that it can omit the resources for fWind.

Now that all the relevant parts of the dialog have been filled in, click OK. After a few seconds you will be returned to the Finder. If you now look in the "Mops *ƒ*" folder, you will find a new icon named "Curves". This is your installed application.

But the application still isn't ready to run, since the menu resources haven't been included in it. This can be done with ResEdit. Start ResEdit and open demo.rsrc, or just double-click demo.rsrc. Do "Select All", then Copy. Then (still in ResEdit), open Curves, and do Paste. Then choose Save to save the updated copy of Curves, and quit ResEdit.

Your new application still won't have a proper icon—it will have just the generic "application" icon—but otherwise it is finished. You can double-click it and run it. We will discuss icons later in part II, chapter 5.

Where to go from here

You've already had quite an exposure to Mops and object oriented programming. You've seen how Mops interacts with the Macintosh Toolbox to simplify the way your programs communicate with the Mac. Now, it's time for you to start experimenting with programs of your own. Several chapters in Part II should point you in the right direction with details of the finer points of Mops programming on the Macintosh.

It is important that you have an acquaintance with the powers of the predefined classes and the words in the Mops dictionary. While there is more to it than a casual reading will ever reveal, you should spend some time studying the methods of the predefined classes as detailed in Part III of this manual to discover what building blocks are available to you. You should also browse through the Mops Index and Glossary in Part IV, where you'll likely discover many built-in words that give you ideas about the operations you can specify for methods.

A vast amount of reference material is available in this manual and in the various Mops files. The best way to make use of it all is to start defining some classes on your own and experiment sending messages to the objects you create. Just as with a spoken language, the more you practice with Mops the faster you'll be comfortable with it.