Part I

A Yerk Tutorial

Table of Contents

Lesson 1

How to Start Up Yerk

To start Yerk, turn on your Macintosh and insert into the internal disk drive slot a working disk containing the the files detailed in the Introduction (if you have an external drive you can put a system disk in the internal drive, and your working Yerk disk in the external drive). If you have a hard disk, copy all of the files over to the hard disk. If you have not made a working disk or copied the files to your hard disk, do so now. Soon after "Welcome to Macintosh" disappears from the screen, the Macintosh desktop appears on the screen with the Yerk disk window open. If the window is not open, double-click the Yerk disk icon. Double-click the icon labeled Yerk.com. After a few seconds, the Yerk window will appear on the screen, as in Figure 1-1.

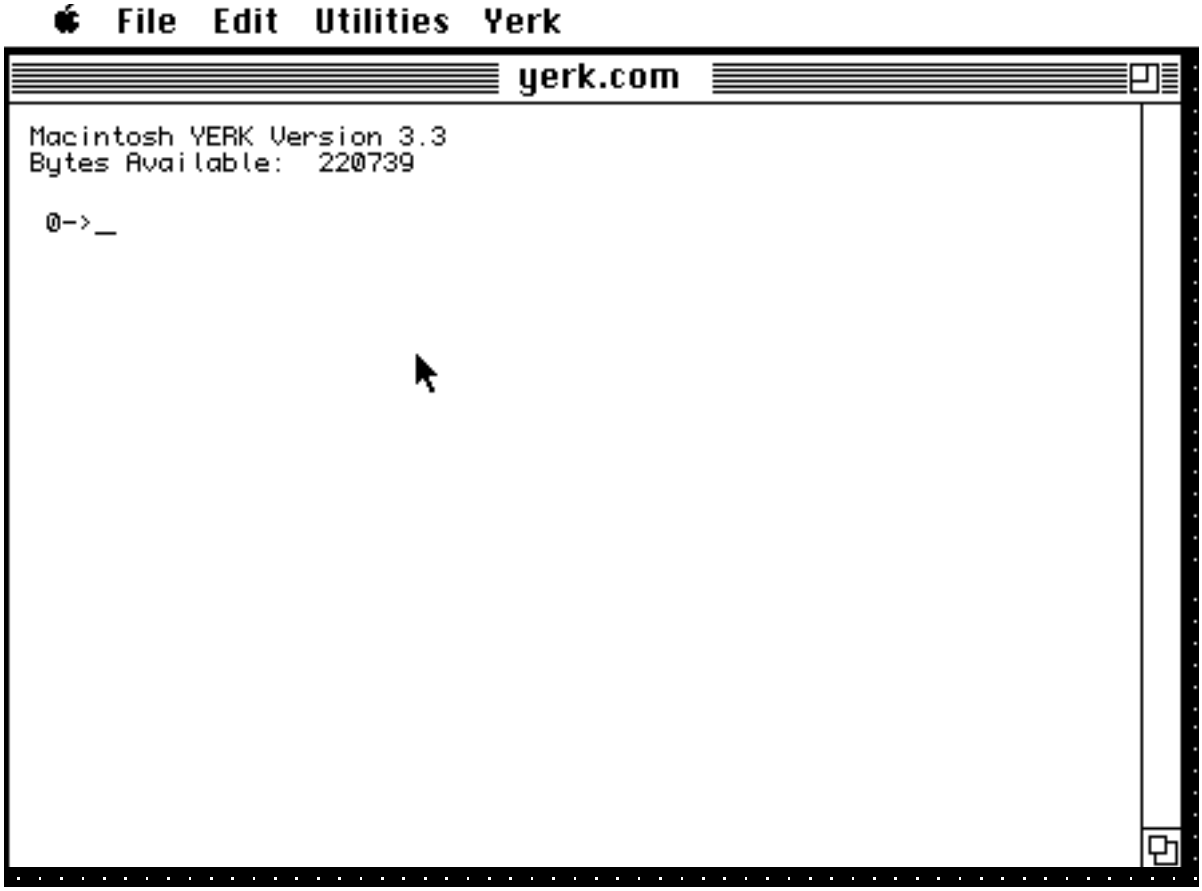


Figure 1-1

Across the top of the screen are 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 a selection to read the copyright and release information about Yerk as well as several desk accessories. An editor should be present as a desk accessory in the

Apple menu. The File menu is much like the File menu in MacWrite and MacPaint, but with a special selection (Load), which you'll use later for loading text files containing your program code.

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

The Yerk Prompt

The starting Yerk window displays the following information:

```
Macintosh Yerk Version 3.64
Bytes Available:   XXXXX
0->_
```

The actual version number of Yerk you will be using may be different -- the version number changes with each update to Yerk. The number of bytes available indicates how much memory is currently available to you for the addition of your program. The values are different depending on how much memory you machine has. If you are using multifinder or System 7, you may modify this amount of memory by selecting the kernel 'yerk' and selecting the 'get info' menu item. Follow the instructions in your Mac manual.

The last line of this display is the Yerk prompt. It consists of the number zero and what looks to be an arrow (the arrow is fashioned out of a hyphen and a greater-than symbol). Press the Return key a couple times. Notice that at each press of the key the prompt moves down one line and appears at the left margin of the screen. Bear in mind for later reference that no matter where along a line the prompt may appear (some graphics commands may end with the prompt somewhere in the middle of the screen), a press of the Return key will bring the Yerk prompt to the left margin of the next line on the screen -- a position that usually makes it much easier to see. If the prompt reaches the bottom of the screen, the prompt line will stay at the bottom of the screen, and all entries above it scroll toward the top of the screen. Pressing the Return key without typing anything else at the prompt will not harm your program in any way.

We said earlier that Yerk behaves like a dictionary. In other words, when you opened Yerk.com just now, the Mac automatically loaded the basic Yerk vocabulary into its memory. Each time you type a word -- any group of text characters -- and press Return, Yerk 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 Yerk dictionary, a message appears on the screen to advise you that Yerk could not find the word. Let's try it.

Type someone's name and press Return:

```
0->michael   <RETURN>
MICHAEL? not found

0->_
```

Notice a few things that happened here. First of all, Yerk beeped to give you an extra warning that something is not quite right. As you do more sophisticated programming with Yerk, you'll discover that the beep, rather than being an annoyance, is a welcome signal to alert you to something amiss in your program.

Second, the message coming back from Yerk questioned the name you typed in. Yerk advises you that the name was not found in the dictionary -- in that split second, Yerk compared the name against nearly 1000 words in the Yerk dictionary.

Third, although you typed the name in lower case letters, Yerk came back to you saying that the name it was checking was in all capital letters. That's because Yerk makes no distinction between upper and lower case letters when it comes to words in its dictionary. Internally, everything is converted to upper case.

And finally, after all the beeps and messages, Yerk restored the 0-> prompt, patiently awaiting the next entry you type in.

Both the zero and the hyphen in the Yerk prompt are significant. While the greater-than symbol never changes, the zero and the hyphen do.

The hyphen indicates which numeric base the computer is in. We'll have much more to say about number bases later, but for now, you should be aware that this part of the Yerk prompt can display one of three characters, depending on the numeric base you wish to work in:

Prompt	Base
->	Decimal (Base 10)
$>	Hexadecimal (Base 16)
?>	Other

Watch what happens when you change the base from decimal (the base that Yerk starts in) to hexadecimal (hex for short). Type "hex" and press Return:

```
0->hex   <RETURN>

0$>_
```

Notice that the dollar sign replaced the hyphen. To change back to decimal, simply type "decimal" and press Return:

```
0$>decimal    <RETURN>

0->_
```

The number before the arrow is actually a counter. It counts how many things are in a section of memory called the parameter stack. When you start Yerk, there is nothing on the stack -- hence the zero at the first prompt. You'll understand where the stack gets its name after we put some numbers in it.

End of lesson 1

Lesson 2

The Parameter Stack

Type the number 7 and press Return:

```
0->7    <RETURN>
1->_
```

Notice that after you press Return, the prompt now shows a one instead of a zero. Type a 3, a space, a 1, and press Return:

```
1->3 1  <RETURN>
3->_
```

The parameter stack counter now reads three, because the 7, 3, and 1 are on the stack. The space you typed between the 3 and 1 told Yerk that you intended those two digits to be two different numbers. If you had typed 31 instead, then the number 31 would have been put on the stack, and the counter would read two. Understanding the way these numbers are stored on the stack is of utmost importance at this stage of learning Yerk.

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.

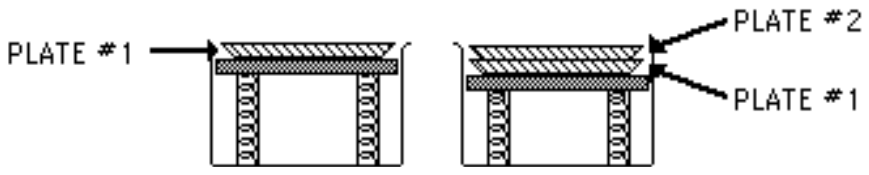


Figure 1-3.

Let's see how this principle applies to the Yerk parameter stack, which you've just loaded with three numbers. If the rule holds, the number 7, which you entered first, should be at the bottom of the

stack, while the number 1, which is the most recent entry, should be at the top of the stack.

To see if these numbers are in that order, take the first available number off the top of the stack. To do this, use the Yerk word that tells the Mac to take the number from the stack and display it on the screen. That word is a simple period (.), called "dot." Type this now.

```
3->.    <RETURN>
1  2->_
```

What happened here was that the dot (print to screen) command pulled the 1 off the top of the stack and displayed it on the screen. The Yerk prompt (2->) now indicates 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 displayed on the screen. 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 a space between each command. If you fail to put the required space there, Yerk thinks that the string of characters is a single Yerk word -- perhaps a word that Yerk cannot find in its dictionary.

To bring the Yerk prompt to the left margin, where it will be less confusing, simply press Return once. Now type two periods, with a space in between, and press Return:

```
2->. .  <RETURN>
3  7  0->_
```

Yerk 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 Yerk 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.

Yerk also has a word, .s (dot s), that displays a list of all numbers on the stack without removing them. To see how it works, place the same three numbers on the stack (don't forget the spaces):

```
0->7 3 1   <RETURN>
3->_
```

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

```
3->.s  <RETURN>
Parameter Stack:
          1  $      1
          3  $      3
          7  $      7
Return Stack:
          16220  $    3F5C
          16678  $    4126
Methods Stack: (--Empty Stack--)
3->_
```

The first grouping lists the contents of the parameter stack (ignore the Return and Methods stacks for now). Importantly, the values are listed such that the number on the top of the stack is shown at the

top of the list to give you a better visual portrayal of the stack's contents. 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 the same hexadecimal indicator as is used for the hex Yerk prompt. In this case, it happens that the Parameter Stack numbers in both bases are the same. Note, too, that the Yerk prompt at the end shows 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 Yerk prompt returns to the left margin of the next line. For example:

```
0->1 10 100   <RETURN>
3->. . . CR   <RETURN>
100  10  1
0->_
```

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

The parameter stack gets its name because a good many operations in Yerk 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 Yerk 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.

Yerk 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 (and Return) as follows:

```
0->7        <RETURN>
1->2        <RETURN>
2->+        <RETURN>
1->.        <RETURN>
```

```
9  0->cr    <RETURN>
0->
```

Let's follow what happened here. You should already understand how the stack counter increments each time you type a number and press Return. In the third line, you type the operation, the + sign for addition. When you press Return, the computer calculates the sum for you. Yerk stores the sum on the stack -- hence the stack counter shows one value on the stack. Note, too, that the original numbers were taken off the stack by the addition operation. For a split instant, there was nothing on the stack, as the two numbers were being added inside the computer. To display the contents of the stack, and the result of your addition, you must issue the dot command. Sure enough, the answer, 9, was on the stack.

Yerk 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:

```
0->7 2 + . cr   <RETURN>
9
0->
```

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 readout of the stack counter. 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 Yerk 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 Yerk word you define in a program should be accompanied by its stack notation. 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 Yerk dictionary. While the notation will at first help you learn how Yerk 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 Yerk, you don't have to type stack notation for words you define at the Yerk 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 Yerk 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, Yerk 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 Yerk 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 Yerk 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 Yerk answer is 3.

```
0->10 3 / . CR   <RETURN>
3
```

```
0->_
```

When you use the divide operation (/) in Yerk, the remainder is lost forever. But Yerk 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! Yerk does not distinguish between upper and lower case).

```
0->10 3 /mod . . cr   <RETURN>
3 1
0->
```

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.

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 YERK's postfix, integer arithmetic environment. For example, you may find yourself confronted with having to include the following formula in a Yerk program:

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

If so, then YERK'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 Yerk 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 Yerk

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

Yerk's Object Orientation

Armed with a basic knowledge of Yerk's stack, you're now ready for an introduction to the language's real power: its object orientation.

If you have experience programming in a procedural language like BASIC and Pascal, Yerk's object orientation may present a challenge at first because it requires an entirely different way of thinking about a program and its execution. Actually, if you have not programmed a computer before, you may find Yerk's object orientation easier to understand the first time through than those who have programmed in other languages.

The next several lessons discuss the concepts of an object oriented language. We will gradually apply the general concepts to Yerk programs in particular. If you are new to object oriented programming, have patience with these lessons. Read them carefully from beginning to end. Because some parts of Yerk are best described in terms of other parts, which may not yet be defined, you may get more from these lessons by reading them a second time.

Fundamental Concepts

The best place to start is to introduce you to the object-oriented components used in Yerk. The primary ones 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 extensive metaphor.

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 1-2.

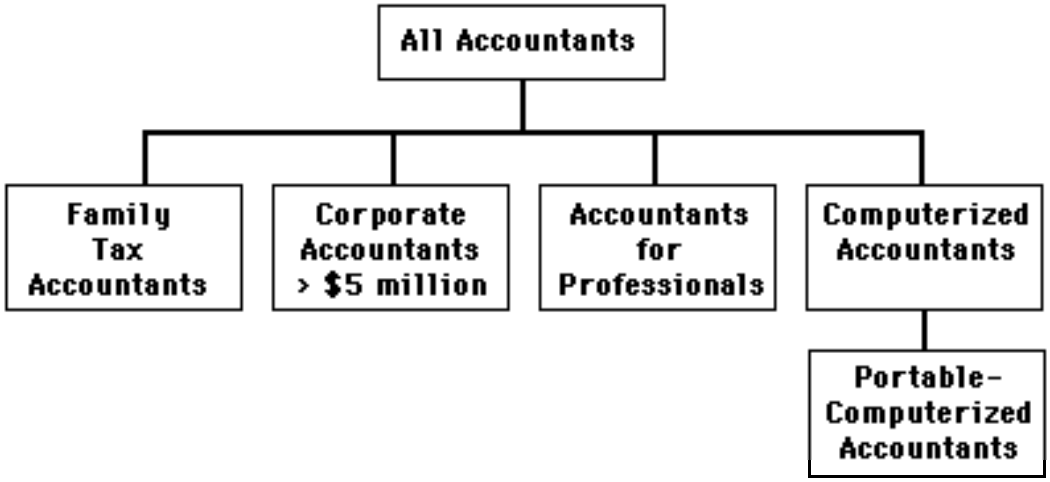


Figure 1-2

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 1-3).

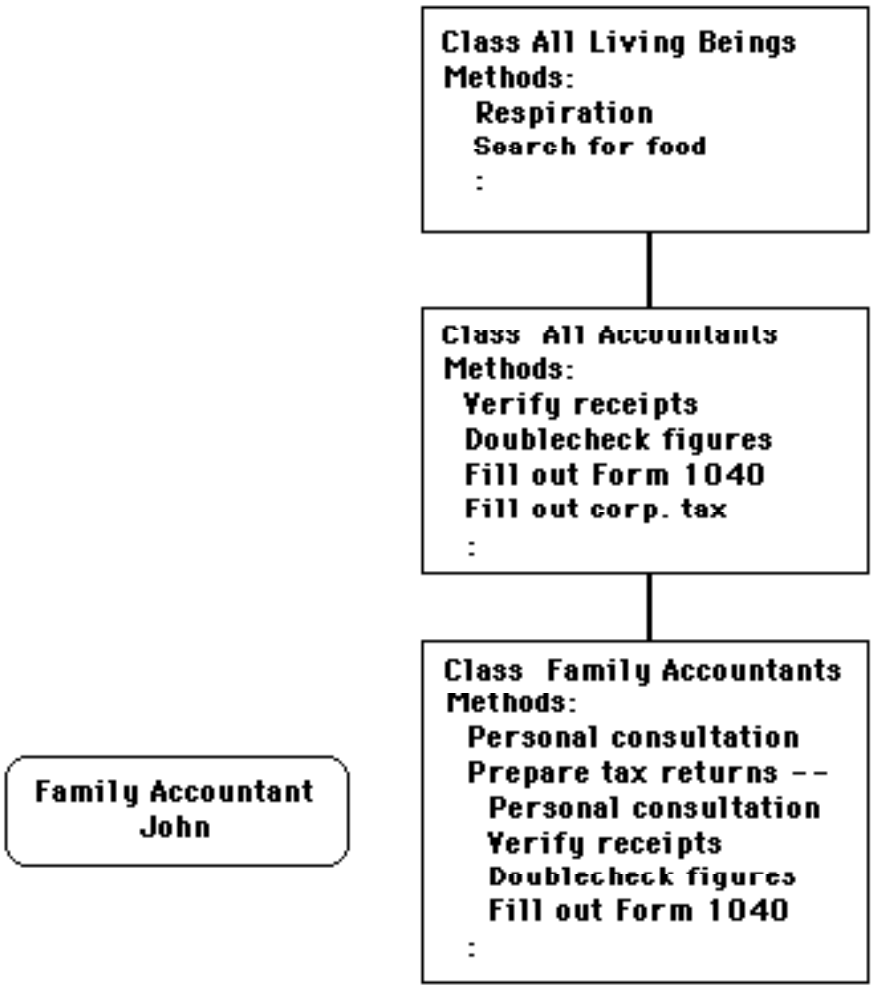


Figure 1-3

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.

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 1-4, 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 1-4.

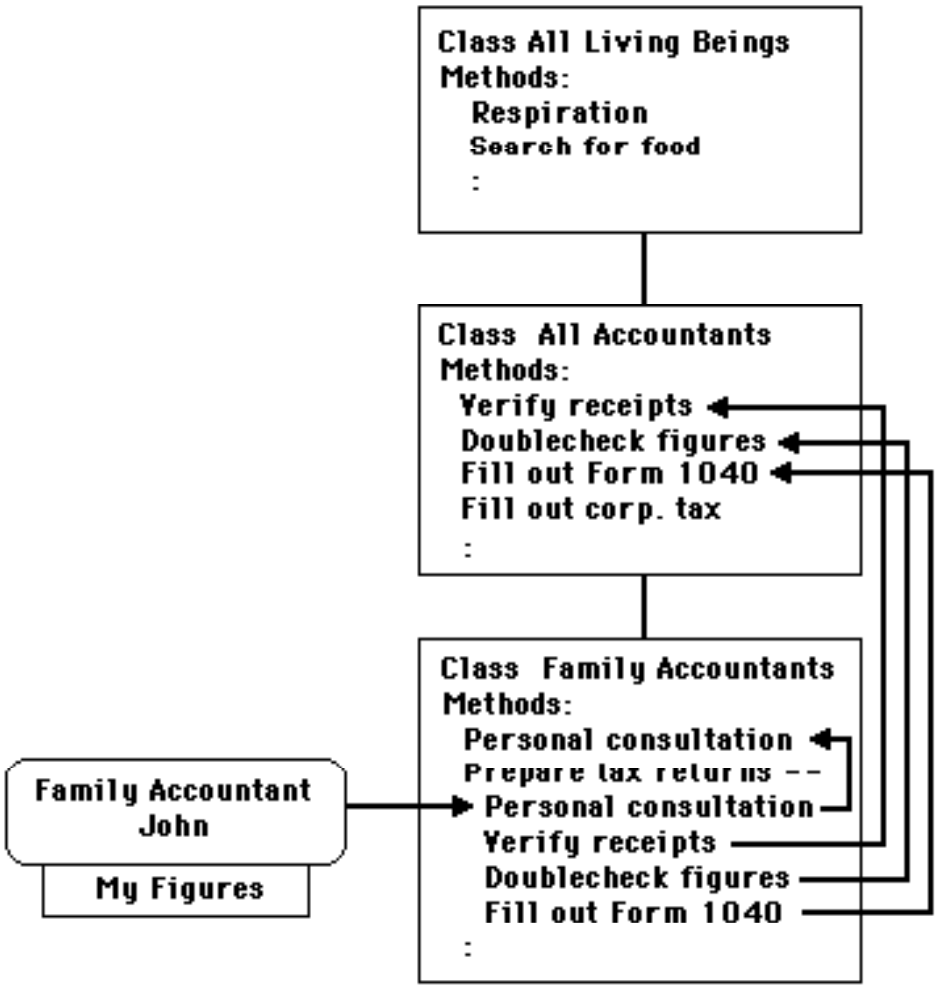


Figure 1-4

When you send the same selector to Percival, but with your mother's figures, Percival follows the same procedures as John, but never see's your figures, which John has to himself (see Figure 1-5).

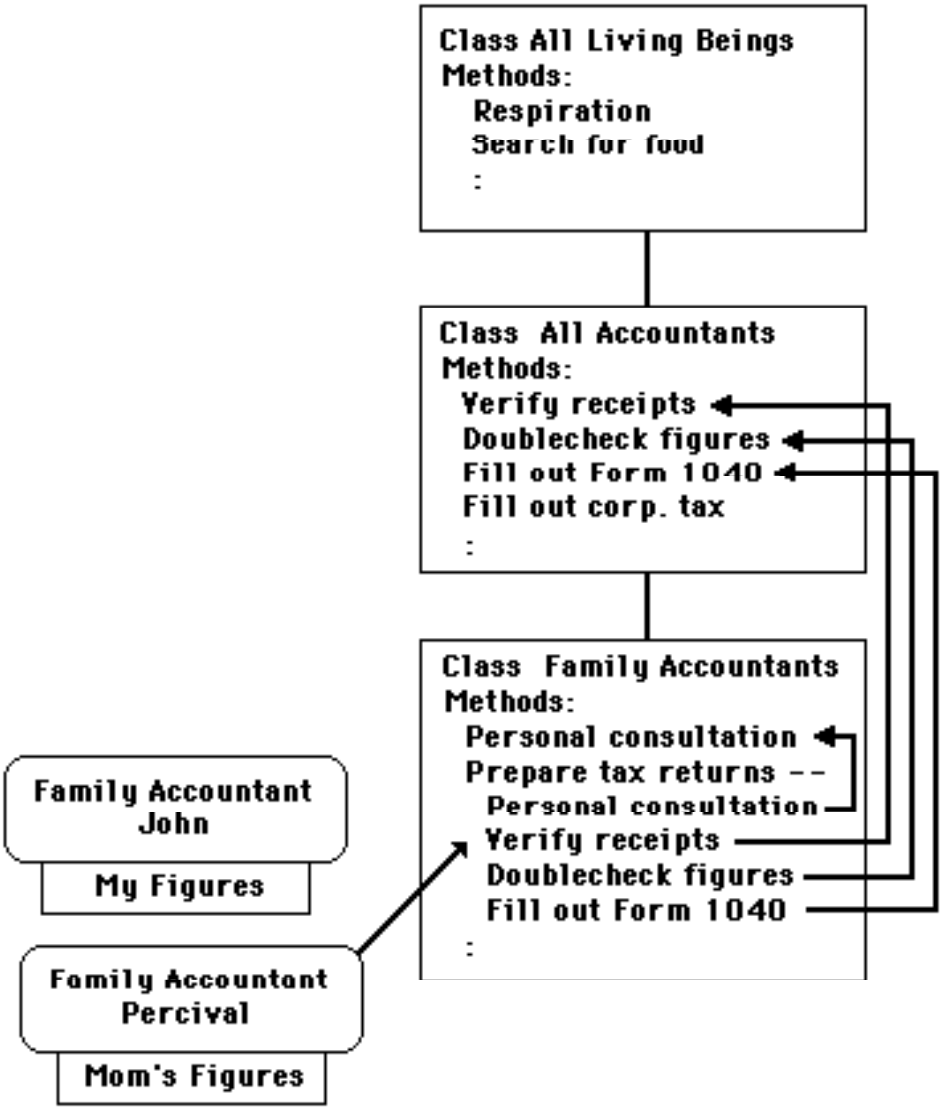


Figure 1-5

End of lesson 4

Lesson 5

Mapping Class-Object Relationships in Yerk

An object oriented language like Yerk 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 Yerk 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 Yerk 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 Yerk 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 Yerk programs really look. Pay particular attention to the physical structure of program listings -- indentions, spacings, capitalizations, and the like. While Yerk is pretty loose about how you make your programs look, the ways prescribed hereafter will help you read printouts of your code for debugging and enhancement. 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 Yerk 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 instance variables (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.

To see the source file containing class Rect, you may use your editor to show the file QD in the System Sources folder; or, you may use Yerk's on-line documenation feature...just enable it by typing

```
0->+docs \ turns on documentation feature
0->see rect
```

A window will appear behind the fwind...you may want to move and resize the fwind if you are going to use the documentation feaure. It contains the QD source scrolled to the beginning of the rect class definition. This window will scroll, but is not yet anything like an editor window. Just use it to quickly see the original source of the definition. You may also use Yerk's decompiler...type:

```
0->de' rect
```

To learn more about these two features, see Part II, chapter 6 on the decompiler.

Returning to the Rect example, the class definition up to this point looks like this:

```
:CLASS  Rect    <Super Object

    Point    TopL
    Point    BotR
```

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." Also, the word "CLASS" is all capitals. While not required, this capitalization will help you find all the classes in a program listing. There are two spaces between :CLASS and the name of the class -- only one is required, but two helps the beginning of the class definition stand out in a listing.

On the same line as the name of the class is a reference to the superclass from which the class Rectangle is derived. Although in this example the superclass name is Object, this should not be confused with Yerk objects. Class Object is a special class that defines the behavior appropriate to all Yerk objects. As such, all classes in Yerk can trace their inheritance to Class Object. 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 Yerk disk as the source file labeled Object) to see what methods are defined in Class Object.

The instance variables tell Yerk 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 TopL[eft] and BotR[ight], both belonging to the Class Point. It would not be possible to create ivars TopL and BotR in Class Rect if Class Point had not been previously defined. Fortunately, Class Point is one of Yerk's many predefined classes.

(For procedural language buffs, a key to understanding the object orientation of Yerk 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 (TopL and BotR) created inside Class Rectangle, you can look up the Yerk source code for the Class Point (located in the qd source file on the Toolbox disk). The class definition looks like this:

```
:CLASS  Point  <Super Object
   INT Y  ( horizontal coordinate )
   INT X  ( vertical coordinate )

 ( x y --  )
   :M  PUT:  Put: Y  Put: X  ;M  ( store points )
       :
       :

;CLASS
```

Right away, you see 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 remarks about the code as a way of documenting the program. Remarks can be placed inside parentheses (with at least one space separating the parentheses from the comment) or preceded by a backslash and a space.

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. The search reveals:

```
:CLASS  INT   <SUPER OBJECT
   2 BYTES DATA

   :M  PUT:  MW!   ;M   ( store integer )
       :
       :

;CLASS
```

Class Int is another one of Yerk's predefined classes. It states, first of all, that its superclass, like many in Yerk, 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).

Going back to the Class Point definition, the method in its fourth line is a single instruction for Yerk

to store both the X and Y coordinates in memory. Therefore, every time one of the ivars (TopL or BotR) 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.

In the method list in the Class Rect definition are two methods:

```
:CLASS  Rect  <Super Object
        Point  TopL
        Point  BotR

        ( l t r b -- )
   :M  PUT:   PUT: BotR  PUT: TopL       ;M
   :M  DRAW:  ABS: 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 BotR reserved cells as soon as the PUT: BotR message finds the definition of the PUT: method in BotR's class, Class Point. The second two integers are placed in the object's TopL cells as the result of the PUT: TopL 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 Yerk 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 absolute address of an object's data. Whereas Yerk addresses are relative to Yerk's starting point in memory, the Toolbox counts addresses only from the beginning of Mac memory. Therefore, the ABS: Self message in the DRAW: method calculates the data's absolute address, which is then passed to the Toolbox call.

Notice in the Class Rect listing how the methods are indented a couple spaces, and the similar parts of the methods statements are aligned vertically. Again, the number of spaces between elements in Yerk is not critical (as long as there is at least one), but messages in methods are easier to read if you separate them by two or three spaces.

To end the class definition, ;CLASS (pronounced "semicolon class") is placed at the left margin. This position helps you pick out the beginning and ending of class definitions on long listings of Yerk source code.

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 Yerk word, "Box1," to the dictionary in memory. You can visualize the object in memory to look like Figure 1-6:

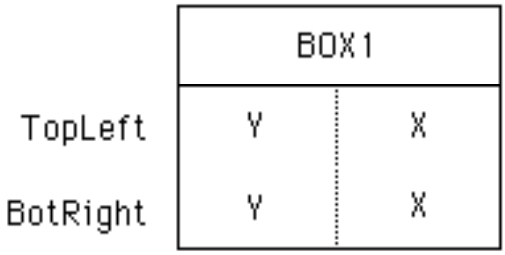


Figure 1-6

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 Yerk 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 TopL and BotR 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
        Point  TopL
        Point  BotR

        ( l t r b -- )    ( store coordinates )
   :M  PUT:   PUT: BotR  PUT: TopL  ;M
   :M  DRAW:  ABS: Self   CALL: FrameRect  ;M

        ( l t r b -- )    ( draw at new coordinates )
   :M  PLACE:  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:

```
300 20 400 100 PLACE: Box1
```

Summary

Before taking one more step, let's summarize. Creating a Yerk 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 1-7. It graphically portrays the relationships between the classes and objects discussed above.

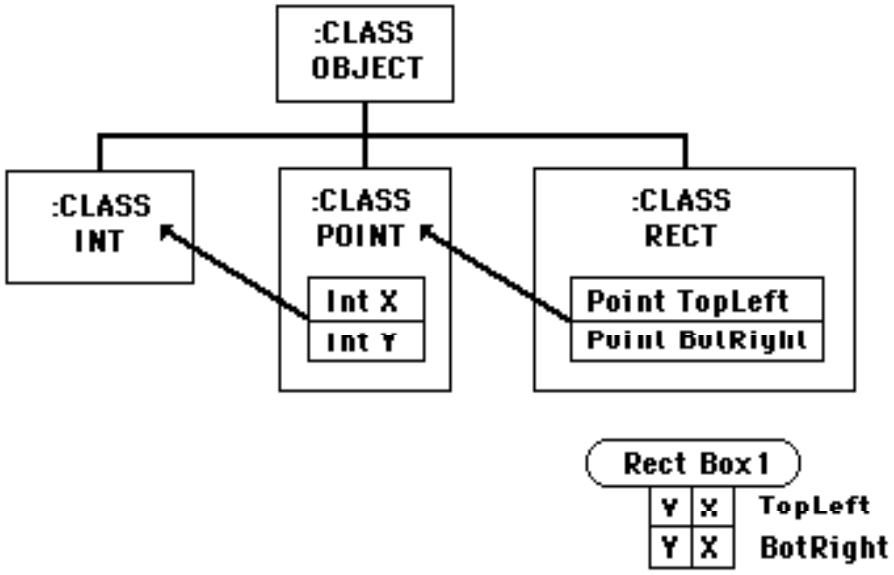


Figure 1-7

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 TopL and BotR; 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 Yerk 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 Yerk 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 Yerk 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 1-8.

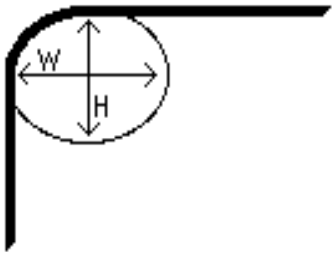


Figure 1-8

The Toolbox call, FrameRoundRect, expects these dimensions as a 4-byte data cell -- a construction that Yerk 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: TopL, BotR, and Ovalsize.

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 (TopL and BotR) 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, (ABS) retrieves the absolute address of the object to be drawn, which in the case is the address of the rectangle coordinates. The Toolbox uses this address

to locate the values it uses as parameters); then 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

        ( w h --   )
   :M  INIT:   PUT:  OvalSize   ;M

        ( -- )
  :M DRAW:  (abs) int: ovalSize call frameRoundRect  ;M

;CLASS
```

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

(Note: If you want to try this yourself, type the above class definitions in an editor, save the source file, and load the file into the yerk.com window. To do this get your editor out. In a clean edit window, type the class definition above. When finished, select Save As... from the Editor menu, and assign a short, recognizable name to the file, like "rr." Close the editor window to return to Yerk.com. Load the file into Yerk.com by selecting Load from the File menu and choosing your file.)

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

```
RndRect Cynthia
```

To draw this object on the screen would take a message like:

```
20 30 init: Cynthia
300 20 400 100 put: Cynthia
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 1-9.

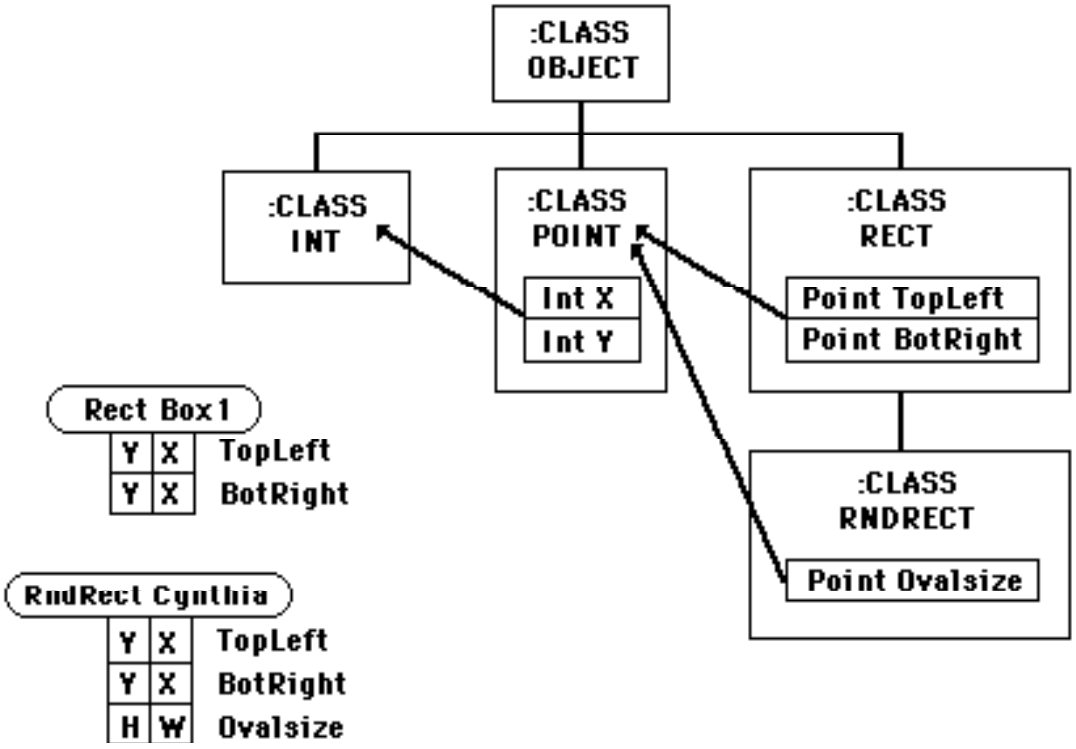


Figure 1-9

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

End of lesson 7

Lesson 8

Predefined Classes -- An Introduction

Yerk 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. Yerk 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 Yerk. 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 Yerk 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 Yerk.com, the text of the source code is also on your Yerk disks. You can view and print these files using either a desk accessory Editor or a word processing program, such as MacWrite. 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 on your Yerk disk. 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 have seen on the Yerk disk, Yerk predefined classes are divided into three groups. One group, called Yerk 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 Yerk.

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 Yerk, and is the Yerk source file called "Object."

Data Structure Classes

Among the most used predefined classes are several that are grouped into a cluster called "data structures." Figure 1-10 shows the organization of the Yerk data structure classes, which are listed in the Toolbox Class file called "Struct" and "Struct1."

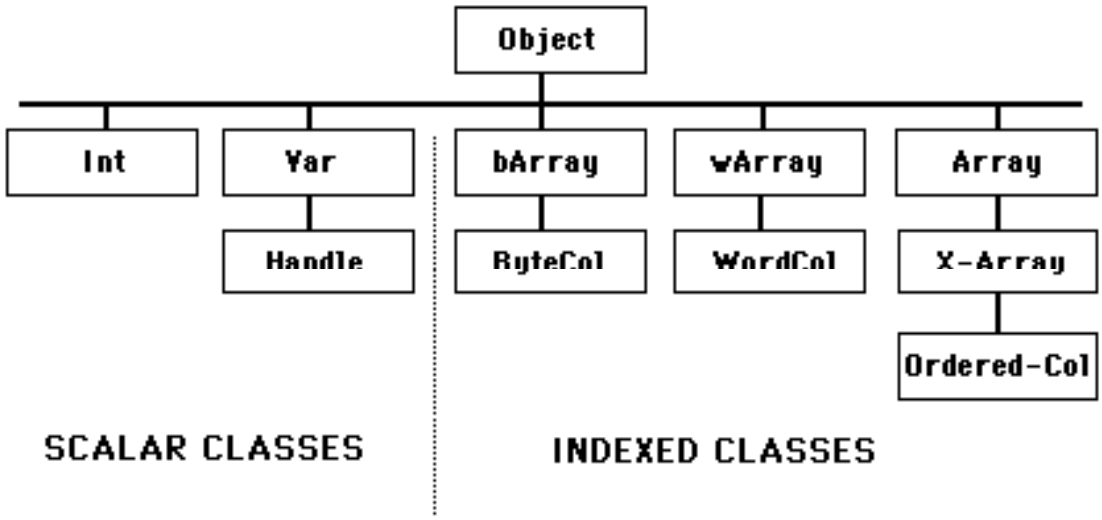


Figure 1-10

These classes form the basis of much number and string (text character) storage and manipulation inside a Yerk 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 three classes to the left of the dotted line in Figure 1-10 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 1-10 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 Yerk 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 1-11 in memory:

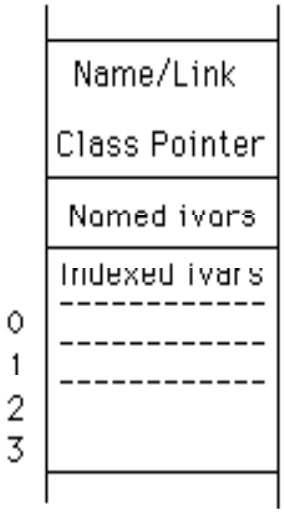


Figure 1-11

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 1-10 include the number of bytes each data cell is to contain (1, 2, or 4) and other considerations discussed later.

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 1-12 shows the QuickDraw classes and the superclasses from which they were derived:

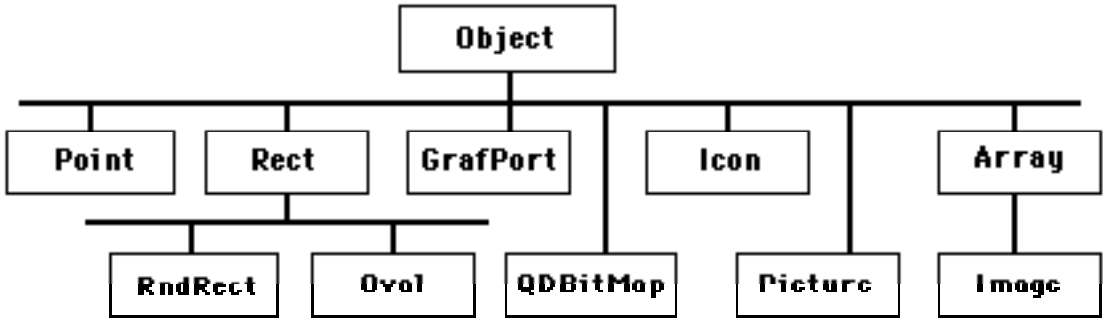


Figure 1-12

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 Yerk'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 Yerk Words

We said earlier that you can add words to the Yerk dictionary while building a program. In fact, that is largely what programming in Yerk is all about. Class names, method names, and object names become part of the dictionary in your program. Defining new words in Yerk 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 Yerk or turn off the computer. 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 select Save As... from the File menu and type in a name for the file. You can also type "Save" at the Yerk prompt, space one space, and type the name you want to give to the file. In either case, you will be able to recall the dictionary at a later time by double-clicking the file icon on the Desktop.

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 Yerk 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 Yerk, a tick is an apostrophe. By typing apostrophe, space, and the word you're testing, Yerk 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:

```
0->' window     <RETURN>
1->.
                <RETURN>
50346  0->       <RETURN>
0->' twindow    <RETURN>

TWINDOW?  not found
File Stack:
Token=TWINDOW

0->_
```

You could, instead of using the "tick" command, use the decompile module in the Utilities Menu, or

type:

0-> **de' window**

<RETURN>

The definition of the word 'window' will be printed in the Yerk window. To stop the listing, hit any key. The decompiler is a handy utility to learn the definitions of words without having to look them up in the source.

You define a new Yerk word by typing a colon, a space, the name of the new word, two spaces (one space is required, the second improves readability in a program listing), 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 Yerk 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 Yerk prompt to the left margin of the next line (remember, if you're typing the definition from the Yerk prompt, you don't need to type the stack definition):

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

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:

```
0->2 6 add   <RETURN>
8
0->
```

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

Named Input Parameters

We're going to make Yerk a little easier for you by reducing what may be undo concern about the way numbers are stored on, and recalled from, the parameter stack. Whenever you define a new Yerk word, Yerk 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 Yerk 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. Whenever the formula is executed like this:

0-> **40 5 12 50 formula <RETURN>**

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. The "solution" parameter is optional, in that it serves no computational purpose, yet it helps readers of your code better understand your definition. 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 Yerk 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 is another operation you can perform to a number stored in a named input parameter. You can add a number to what is there with the ++> operation. 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
             2  c  *  b  +        -> den
             num  den  /            ;
```

In this example, a, b, and c in the curly brackets are named input 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.

When you use named input parameters and local variables in the same definition, your worries about the stack nearly disappear.

End of lesson 9

Lesson 10

Additional Math

This is a good time to learn several other Yerk 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) and n2 on the stack.	Leaves the smaller of n1
MAX	(n1 n2 -- n-max) and n2 on the stack.	Leaves the larger of n1

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

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

1+	(n -- n+1) the stack.	Adds 1 to the number on
1-	(n -- n-1) number on the stack.	Subtracts 1 from the
2+	(n -- n+2) the stack.	Adds 2 to the number on
2-	(n -- n-2) number on the stack.	Subtracts 2 from the

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 -- n+4)	Adds 4 to the number on the stack.
4*	(n -- 4n)	Multiplies the number on the stack by 4.
8+	(n -- n+8)	Adds 8 to the number on the stack.

The application of these shortcuts will become more apparent the more you program in Yerk. The addition and subtraction shortcuts, for example, come in handy when you need to increment or decrement a counter of some kind.

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 Yerk, the simplest way to display a text message is by preceding it with a special print command called the dot-quote, or ." in Yerk notation. It's just like the dot command, but instead of looking to the stack for something to display, the dot-quote command displays the element that follows the quote, up to a closing quotation mark. The quotation marks fall into a broad category of symbols in computer languages called delimiters, because they set the limits of a given operation -- in this case the display of a canned message. The text message inside delimiters is called a text string, or just string.

Text strings can be made part of Yerk 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 Yerk operating on the Macintosh. "  cr ;
```

Now, when you type "hi" at a Yerk prompt, the message between the quotes appears on the screen in capital letters.

One of the nice things about Yerk is that you can use previously defined words inside the definitions of new words. Therefore, you could take the "hi" Yerk word and incorporate it inside yet another Yerk 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 Yerk 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. Yerk will alert you that you have redefined the word when you press Return. Here's the new definition:

```
: add  ." The sum is: " + . cr ;   <RETURN>

ADD is redefined   0->_
```

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

```
0->10 20 add   <RETURN>
The sum is:  30
```

0->

Explicit Stack Manipulations

While named input parameters and local variables will disguise most stack manipulation for you, there may be instances in the development of a Yerk program 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 rare cases, you can choose from a series of stack manipulation commands which should get you out of the stickiest problems.

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:

```
( denom num1 num2 num3 -- solution )
: formula * * 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 Yerk word "cubed" to perform the cube calculation:

```
( n1 --  )
: cubed dup dup * * . cr ;
```

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

```
0->3 cubed   <RETURN>
```

```
27
0->_
```

Experiment with the other stack manipulation operators described above. Place a few numbers in the parameters stack and issue the commands. Then display the contents of the stack (remember .s) to see exactly how the contents of the stack are affected by those operators. If you need to, you can combine two or more stack manipulation operators in the same Yerk word definition as your arithmetic needs arise.

But overall, named input parameters and local variables are the preferred way of handling numbers on the Yerk 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 Yerk 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 Yerk, the IF...THEN decision process is a bit different from some other languages you may know, largely because of Yerk's 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 Yerk 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 Yerk 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 as easily as the operations you learned so far. That's because this construction must be compiled before it will run on Yerk. Fortunately, there is a simple way to compile an IF...THEN statement without having to write a line or two of code in an Editor and load it into Yerk (which would be somewhat time consuming for the simple purpose of experimentation). Instead, you can put an IF...THEN statement inside a colon definition (the contents of a colon definition are compiled when you press Return after the semicolon delimiter). Type the following:

```
( n --  )
```

: **test <RETURN>**
IF ." It's true there is a non-zero number on the stack. " <RETURN>
THEN cr ; <RETURN>

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" warning message to appear. A zero, on the other hand, is indeed a number, and it occupies space on the stack.

Notice, too, how the construction is organized. The control words of the process, IF and THEN, are indented, capitalized, and aligned with each other. This is done for the sake of readability when looking over a long program listing. It helps you spot the pieces of the construction much more quickly. Pay close attention to the formatting in all our examples, and follow the formats whenever possible.

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 Yerk. 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. When you type a redefinition in the Yerk window and press Return after the word, Yerk leaves the message that the word is redefined, but the full definition won't be complete until you type the semicolon:

```
: test   <RETURN>

TEST is redefined IF   ." Non-zero number on stack "
          <RETURN>
               ELSE ." Zero on stack "          <RETURN>
               THEN  cr ;                       <RETURN>
```

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

0->**1 0 3 <RETURN>**
3->**test <RETURN>**
Non-zero number on stack
2->**test <RETURN>**
Zero on stack
1->**test <RETURN>**
Non-zero number on stack

As with nearly all Yerk 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, Yerk adheres to a programming language convention revolving around the terms TRUE and FALSE. These words are Yerk 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. In most cases, however, when a comparison operation returns TRUE (non-zero) to the stack, the number placed there is a 1. Similarly, when a comparison operation returns FALSE to the stack, the top number on the stack is a zero.

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.

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 one, zero, and four in the stack and run the test three times:

```
0->1 0 4   <RETURN>
3->test    <RETURN>
True
2->test    <RETURN>
False
1->test    <RETURN>
True
0->_
```

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

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 Yerk, you simply move the operation sign to the right:

n1 n2 >

But in this case, Yerk 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)

IF1  xx
     IF2  ww
          IF3 uu
          ELSE zz
          THEN3
     THEN2 qq
THEN1 yy
```

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.

YERK'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.

End of lesson 11

Lesson 12

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.

Both of these operations look at the binary makeup of two numbers and perform binary arithmetic on them to determine the state (TRUE or FALSE, 1 or 0, on or off) of each bit in their binary equivalents.

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

In Yerk, 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 Yerk 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:

```
0->1 3 AND . cr   <RETURN>
```

```
1
0->_
```

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 Yerk to do these operations correctly for you at all times.

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.

Yerk 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 Yerk 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:

```
  ( n1...n10 -- sum )
: addten  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 Yerk 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 Yerk's powerful features, called

indexing, which will play an increasingly important role the more you learn about Yerk.

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 Yerk word that copies the index to the parameter stack is:

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

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

```
 ( n -- )
: 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 Yerk 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  0 10
             DO  i  .  -1
             +LOOP  cr  ." 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.

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:

```
30000 0 DO LOOP
```

after the dot statement inside the original DO...LOOP operation. Perhaps a better way would be to define a new word, "delay," to handle this timing delay in not only the loop (between the counting down) but between the ignition and liftoff. Define "delay" as:

```
: delay 30000 0
        DO
        LOOP   ;
```

Each time "delay" is encountered in the program, the computer appears to mark time while the DO...LOOP construction whizzes in circles 30,000 times. Now redefine "countdown" as:

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

Type "countdown" and watch the seconds tick away:

```
0->countdown   <RETURN>
10  9  8  7  6  5  4  3  2  1
Ignition
Liftoff
0->_
```

Incidentally, if you need access to the index of a nested loop, Yerk has a predefined word that allows you to copy that number to the parameter stack, just like "I" copies the outermost loop index number to the stack. That word is "J."

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

In other words, "J" looks up the index of the loop just inside the outermost DO...LOOP construction, and copies the current number to the parameter stack.

Indefinite Loops

An indefinite loop is another kind of loop you'll use from time to time in a Yerk 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 Yerk, 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.

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 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 tap 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 TRUE flag appears on the stack; loop ends when stack shows FALSE flag.

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

When you're designing loops, it is sometimes possible for an infinite loop to slip in accidentally. 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 stop the loop. Otherwise, your program will appear to "lock up" and be unresponsive to your keyboard input. If an infinite loop slips into your program, there is nothing you can do except to turn off the computer or press the INTERRUPT button of the Macintosh Programmer's Switch, which you should install on the left rear side of your Mac. When you press the INTERRUPT button, a warning box appears on the screen, allowing you to resume inside Yerk.

End of lesson 12

Lesson 13

Yerk's Fixed-Point Arithmetic

The basic version of Yerk (Yerk.com) utilizes fixed-point arithmetic, also called integer arithmetic instead of floating-point arithmetic (YerkFP.com). 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 Yerk: 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 Yerk programmer

Foremost is that floating-point arithmetic takes up more memory in the computer, increasing the size of the Yerk 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 the computer and the language, a floating-point calculation can take up to three 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 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 Yerk 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, Yerk 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 Yerk arithmetic operations. But the order allows you to better visualize the process by changing the algebraic infix notation of a problem to Yerk 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 Yerk 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), Yerk 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 Yerk 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 Yerk that you intend to use a signed number. If, on the other hand, you were to enter the number three billion onto the stack, Yerk 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, and issue the dot command.

```
0->3000000000  <RETURN>
1->. cr  <RETURN>
-1294967296
```

```
0->_
```

Sure enough, the result printed as a signed number equivalent, a negative number near 1 billion.

Conversely, let's roll back that imaginary tape counter and 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 prints it to 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:

```
0->-1   <RETURN>
1->U. cr   <RETURN>
4294967295
0->_
```

Here are the other unsigned operations found in Yerk:

U*	(u1 u2 -- ud)	Multiplies two unsigned single-precision numbers and leaves their unsigned double-precision product on the stack.
U/	(ud u1 -- u2 u3)	Divides u1 (unsigned single-precision number) into ud (unsigned double-precision number), and leaves their unsigned single precision remainder (u2) and quotient (u3) on the stack.
U<	(u1 u2 -- boolean)	Compares two unsigned single-precision numbers. If u1 is less than u2, then leaves a TRUE flag on the stack; otherwise, leaves a FALSE flag.
U>	(u1 u2 -- boolean)	Compares two unsigned single-precision numbers. If u1 is greater than u2, then leaves a TRUE flag on the stack; otherwise, leaves a FALSE flag.

The above definitions talk of single- and double-precision numbers. In Yerk the distinction between the two is between 32 and 64-bit numbers. Single-precision numbers are stored as 32-bit binary numbers, while double-precision numbers are stored as 64-bit binary numbers. Double-precision numbers are used not so much for large numbers (which are almost always taken care of by the four billion number range of the single-precision number), but for handling parameters which must be passed to various Macintosh Toolbox operations, about which you'll learn later.

Be aware, however, that there are Yerk statements that help you manipulate double-precision numbers. They are:

D+	(d1 d2 -- dsum)	Adds two double-precision numbers, leaving their double-precision sum on the stack.

D.	(d --)	Performs a binary-to-ASCII conversion on a signed double-precision number, and displays the results on the screen.
D<	(d1 d2 -- boolean)	Compares two signed double-precision numbers. If d1 is less than d2, leaves a TRUE flag on the stack; otherwise, leaves a FALSE flag.
D=	(d1 d2 -- boolean)	Compares two signed double-precision numbers. If d1 is equal to d2, leaves a TRUE flag on the stack; otherwise, leaves a FALSE flag.
D>	(d1 d2 -- boolean)	Compares two signed double-precision numbers. If d1 is greater than d2, leaves a TRUE flag on the stack; otherwise, leaves a FALSE flag.
DABS	(d -- d)	Leaves the absolute vale of a signed double-precision number on the stack.

It is important to remember that a double-precision number is stored on the stack in two parts, called the most and least significant 32 bits (also called the high and low longwords). The most significant bits are the leftmost 32 bits of an entire 64-digit binary number. These 32 bits are stored on the top of the stack whenever a double-precision number is on the stack. To demonstrate how important it is to know which half is on top, type the following two series:

```
0->1 0 d. cr  <RETURN>
1
0->_
```

and

```
0->0 1 d. cr  <RETURN>
4294967296
0->_
```

As the D. command assembles a number out of the high and low bits stored in the top two position of the stack, it does so by placing the topmost number as the high bits. In the first example, the top of the stack contained zero, while the second number was a one, recreating the binary number:

0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0001 = 1

But in the reverse order, the 1 becomes the last digit of the high order bits, as in:

0000 0000 0000 0000 0000 0000 0000 0001 0000 0000 0000 0000 0000

0000 0000 0000 = 4,294,967,296

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 shortcut Yerk word that displays on the screen the character that is referenced by its ASCII number. Its definition is as follows:

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

Other Yerk words that might go along with EMIT are:

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

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. But Yerk 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:

```
0->25 value Jane  <RETURN>
```

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 parameter stack. Type:

```
0->Jane   <RETURN>
1->
```

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 the number stored in a value name with the ++> operation, like this:

```
13 ++> Jane
```

This adds the number 13 to the 37 that is already stored there. Decrementing the number in Jane simply requires that you increment 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

:

:

How Yerk Remembers Definitions

Near the beginning of this manual we said a Yerk program builds a dictionary of words. Each word and its definition occupies a portion of the computer's memory, the amount of which depends on the complexity of the definition. If you visualize the Mac's memory as a tall column of one-byte-wide cells, then the words and their definitions would look as if they were stacked atop one another. Each cell has a locator number associated with it, called an address. A Yerk program makes substantial use of addresses behind the scenes in a running program.

But each word on that column of words contains much more information than just its name. A Yerk definition consists of several parts, including information like whether a word is a value or a colon definition, the numbers or other data associated with the word, and where in memory the computer can find the definitions of words used to define that word. In Yerk, the spaces reserved for those parts of a Yerk definition are called fields. Although you won't be working much with the actual fields in your programs, it is helpful to understand how they work and what the terminology means if you encounter it later.

Most Yerk words have four fields that you should be aware of:

name field
link field
code field
parameter field

and they look like Figure 1-13 on the memory column:

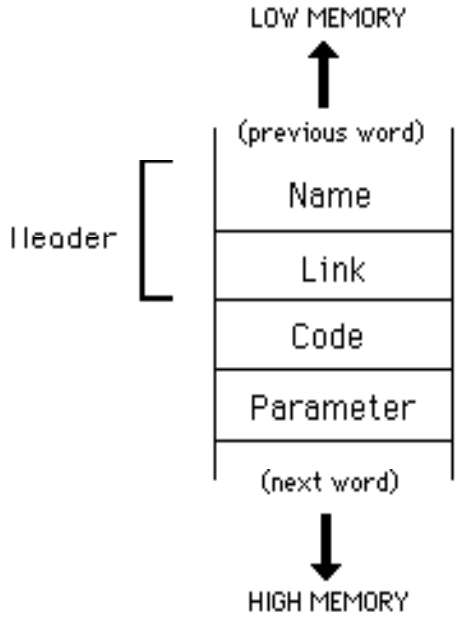


Figure 1-13

The name and link fields are usually grouped together as the header field. The Name field holds the actual name you assign to a colon definition, variable, and so on. It's length varies with the length of the actual name.

The Link field is important to Yerk because it helps Yerk programs run fast. In the link field is the address of the next previous word in the dictionary. This facilitates the search through the dictionary each time you type a previously defined word. The search starts at the most recently defined word (the word nearest High Memory). If there is no match in the first word, the search looks to the link field for the address of the next word on the list and so on backward through the dictionary. The length of the name and parameter fields can change from definition to definition so there is not a fixed memory interval between words.

The content of the Code field specifies whether the word is a colon definition, a value, and so on. In the code field of a Yerk word is a memory address. During execution, the code field is saying to the Mac, "Look at the instructions located in memory address XXXX to see what this definition is." Or, in other words, the address in the code field points to another memory address where specific instructions can be found. At the memory address specified by the code field address (cfa) are instructions that make a value a value. Later, you'll also see that the cfa points to other kinds of instructions, but the principle still holds true: the cfa is a pointer to further instructions that determine what kind of word that particular Yerk dictionary entry is. Yerk dictionary entries without cfas would be like a Webster's dictionary without the abbreviated notations for noun, verb, adjective, etc.

The cells of the parameter field contains the actual data associated with a word. That data can be raw numbers, as in a number associated with a constant. Data can also be pointers (addresses) to other words, as you'll see in a moment. Therefore, some parameter fields consist of only a single cell, while others can be dozens of cells long: it depends on what kind of Yerk word is being defined. But no matter how long the list of parameters is, the cell containing the first piece of data has an address called the parameter field address (pfa) for that particular word.

A good example of parameter fields filled with pointers is a colon definition. A colon definition fills its parameter cells with the addresses of the words inside its definition. For example, if you type in the following definition,

```
: formula  swap dup * + . cr ;
```

the parameter cells in the "formula" word in memory contain the addresses of each word in the definition, seven addresses (including the semicolon) in this case. The addresses in the parameter cells point to the cfas of each defining word, like SWAP and DUP. As you might imagine, there is a lot of pointing going on during the execution of a Yerk program.

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 Yerk programs work.

End of lesson 14

Lesson 15

One of the best ways to learn the fine points of Yerk programming is to study existing programs and then work slowly to customize them by modifying methods, defining new subclasses, creating new Yerk 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 Yerk 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 Yerk code, as the Sin and Turtle files will reveal when you look at them with an Editor or with MacWrite.

The Programs

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 Yerk 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 1-14,

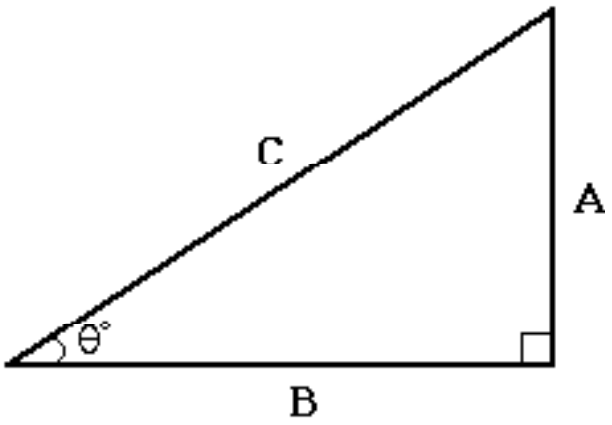


Figure 1-12.

Figure 1-14

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 1-15).

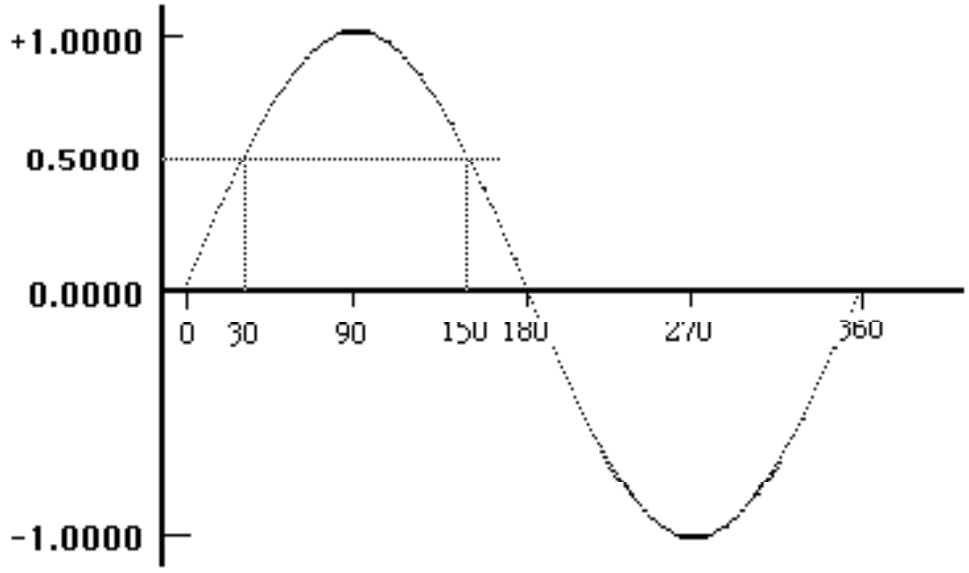


Figure 1-13.

Figure 1-15

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 Yerk 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               ( Sin - sine table )
2
3
4               :CLASS TrigTable  <Super wArray
5                  4 bArray    Signs       (1 if negative in quadrant, 0 if positive )
6                  4 Array     AxisVals    ( 90° values)
7
8                   ( deg -- sin )  ( Lookup a sin * 10000 of an angle)
9               :M  SIN: { degree \ quadrant -- sin }
10                      degree 360 mod          ( put degree in range 0 to 359 )
11                      degree  0<    ( test for negative degree )
12                      IF
13                          dup         ( test that negative degree not multiple of -360)
14                          IF
15                              negate 360 +    ( invert to equivalent positive degree)
16                          THEN
17                      THEN
18                      90 /mod    ( convert degree to range 0-89 and get quadrant)
19                      -> quadrant -> degree
20                      degree 0=     ( test for an axis )
21                      IF
22                          quadrant At: AxisVals   ( if an axis, get value )
23                      ELSE
24                          quadrant 1 and      (true for mirror quadrants 1,3 )
25                          IF
26                              90 degree -       ( create mirror image)
27                          ELSE
28                              degree
29                          THEN
30                          At: Self     ( get sin for this degree)
31                          quadrant At: signs  ( get flag for negative sin)
32                          IF
33                              negate
34                          THEN
35                      THEN         ;M
36
37                      ( deg -- cos)
38                      :M  COS:  90 + Sin: Self ;M   ( cos is sin shifted by 90 °)
39
40                      :M  CLASSINIT:  0 0 To: Signs   0 1 To: Signs
41                          1 2 To: Signs   1 3 To: Signs
42                          0 0 To: AxisVals    10000 1 To: AxisVals
43                          0 2 To: AxisVals   -10000 3 To: AxisVals ;M
44              ;CLASS
```

```
45
46              90 TrigTable Sines  ( system-wide table of sines)
47
48              (  val  angle -- )  ( Fill a Sin table entry)
```

```
49              : 's   To: Sines  ;
50               00 's  00175 01 's 00349 02 's 00524 03 's 00698 04 's
51               05 's  01045 06 's 01219 07 's 01392 08 's 01571 09 's
52               10 's  01908 11 's 02079 12 's 02250 13 's 02419 14 's
53               15 's  02756 16 's 02924 17 's 03090 18 's 03256 19 's
54               20 's  03584 21 's 03746 22 's 03907 23 's 04067 24 's
55               25 's  04384 26 's 04540 27 's 04695 28 's 04848 29 's
56               30 's  05150 31 's 05299 32 's 05446 33 's 05592 34 's
57               35 's  05878 36 's 06018 37 's 06157 38 's 06293 39 's
58               40 's  06561 41 's 06691 42 's 06820 43 's 06947 44 's
59               45 's  07193 46 's 07314 47 's 07431 48 's 07547 49 's
60               50 's  07771 51 's 07880 52 's 07986 53 's 08090 54 's
61               55 's  08290 56 's 08387 57 's 08480 58 's 08572 59 's
62               60 's  08746 61 's 08829 62 's 08910 63 's 08988 64 's
63               65 's  09135 66 's 09205 67 's 09272 68 's 09336 69 's
64               70 's  09455 71 's 09511 72 's 09563 73 's 09613 74 's
65               75 's  09703 76 's 09744 77 's 09781 78 's 09816 79 's
66               80 's  09877 81 's 09903 82 's 09925 83 's 09945 84 's
67               85 's  09976 86 's 09986 87 's 09994 88 's 09998 89 's
68
69              : Sin  Sin: Sines ;
70              : Cos  Cos: Sines ;
71
72              :CLASS Angle  <Super Int
73                      :M  SIN:   Get: Self Sin   ;M
74                      :M  COS:   Get: Self Cos   ;M
75              ;CLASS
```

Building a Sine Table

Let's start with the Sin source code, which is numbered from lines 1 to 75.

Line 1

This is a comment that serves as a plain English heading for the source code, describing what this module does. This particular module creates a table of sine values that Turtle will use to draw complex curves and graphics. Notice the syntax of the comment: open parenthesis, a space, the comment, a space, and close parenthesis. The space between the open parenthesis and the beginning of the comment is critical, because it sets off the lines as a comment. Comments are not compiled into the final code, so be as liberal with comments throughout the program as possible to make it easier for you and others to reconstruct the execution of program parts later on.

Line 4

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 25-46). 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 Array (in the struct file), you'll notice that wArray is defined as an indexed class:

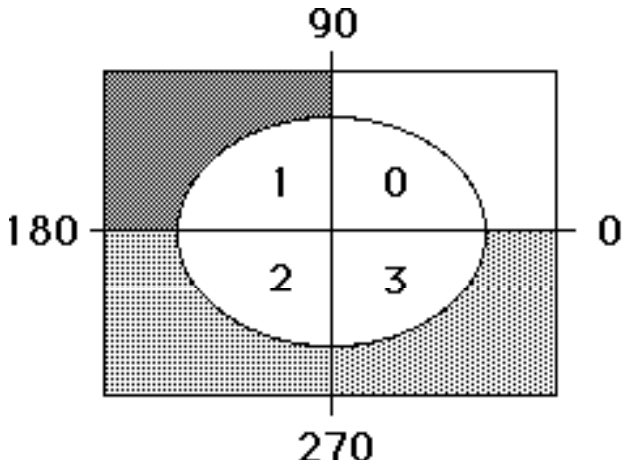
:CLASS wArray <SUPER Object 2 <indexed

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 4 in the class Array definition indicates that each slot is to be 4 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 25, 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.

Lines 5-6
These two lines establish the named instance variables for an object of Class TrigTable. Every object created from Class TrigTable will have space reserved for the two arrays created here, as well as the indexed data noted above. The two arrays are preceded by the number of elements that they will contain in every instance of class TrigTable, 4 in both cases.

The first array, Signs, is an array that will be storing a boolean flag to indicate whether a sine value is positive or negative, depending on which quadrant the value is located (more on this in a moment). Since a boolean flag only occupies 1 byte, Signs is declared to be an instance of bArray which has 1 byte wide elements (the source for bArray is also in struct1).

The other 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 1-16 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 1-16

Line 8
On the left is the stack notation for what takes place in the execution of the following method, SIN:. The stack notation 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.

The comment to the right of the stack notation 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.

Line 9
This begins the definition of the method SIN:, which, as the stack notation and comment in the previous line indicate, converts degree integers into sine values. This line is also where the named input parameters and local variable for this method are detailed inside the curly brackets. References to the 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. The parameter passed to this SIN: method from a message will be assigned to the name "degree." 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.

Lines 10 - 35
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. 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 40 - 43

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 eight messages, all of them TO: operations. The TO: selector of these messages is defined by a TO: method in the receiver's class, bArray for Signs and wArray for AxisVals.

In the first four TO: messages inside CLASSINIT:, the storage will be to the instance variable Signs, which, as noted at the beginning of the current class, was set up as an array in a TrigTable object to hold four values (we intend to have only one TrigTable object, so we won't be duplicating these arrays unnecessarily in other objects of the same class). Now notice the parameters that precede the TO: selectors of these four messages. If you read the messages from left-to-right, you'll notice a pattern: the rightmost value of each pair increments by one, running from zero to three. These values are the index numbers which the array will use to label each of the four actual values that are stored in the array. The values to be stored in the array, again reading from left-to-right, are 0, 0, 1, and 1. In other words, the Signs array inside the TrigTable object could be visualized as something like Figure 1-17:

Index	Data
0	00
1	00
2	01
3	01

Figure 1-17

Likewise, the next four TO: messages store indexed values into the array called AxisVals.

Since Class TrigTable has now been defined (all the code from line 14 through line 23), we can now create an actual table in memory as an object of that class. The statement in line 46 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 48 - 67

While the columns of numbers in lines 50 through 67 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. Line 48 shows the stack notation for the Yerk word definition that occurs in line 49. The definition defines a Yerk 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 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 48 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. The hex values for the Sines array start filling up the object's memory, and look like Figure 1-18:

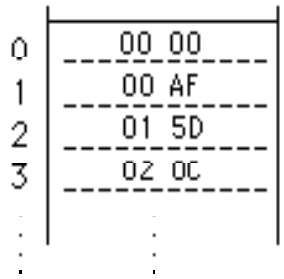


Figure 1-18

What Happens On the Stack

Now we can go back to Method SIN: in lines 9 to 35 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
degree	35	180	293
	35	180	293
0<	0	0	0
	35	180	293
IF	35	180	293
dup	:	:	:
IF	:	:	:
negate	:	:	:
360	:	:	:

+	:	:	:
THEN	:	:	:
THEN	:	:	:

90	90	90	90
	35	180	293
/mod	0	2	3
	35	0	23
-> quadrant	35	0	23
-> degree	---	---	---
degree	35	0	23
0=	0	1	0
IF	---	---	---
	:		:
quadrant	:	2	:
	:		:
At: AxisVals	:	0	:
ELSE	---	:	---
quadrant	0	:	3
1	1	:	1
	0	:	3
and	0	:	1
IF	---	:	---
	:	:	
90	:	:	90
	:	:	
degree	:	:	23
	:	:	90
	:	:	
-	:	:	67
	:	:	:
ELSE	---	:	:
		:	:
degree	35	:	:
		:	:
THEN	35	:	67
		:	
At: Self	5736	:	9205

		:	
quadrant	0	:	3
	5736	:	9205
		:	

At: Signs	0	:	1
	5736	:	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 10 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 positive value between 0 and 359, and further normalization is required. Line 11 tests the degree entry to see if it was negative. If it was negative, lines 12 - 15 perform the additional normalization. Lines 13 - 14 perform a test to see if the partially normalized result is zero, in which case the value is alright as is. If it is not zero, then line 15 puts it correctly in the range 0 to 359.

The /mod operation on line 18 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 19 these values are taken from the stack and put into local storage.

For the next operation on line 20, 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 IF statement on line 21 is performed. On line 22 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 fetches 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 26 (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 28.

In line 30, 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 Yerk 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. To check this, the sine value and quadrant number are swapped. The quadrant number is used as an index to the Signs array in another AT: fetch operation (line 31). The values in that array are either 1 or 0, depending on whether the quadrant requires a negative or positive sine value, respectively . If the value is a 1, then the sine value, which is all that remains on the stack, is made negative (with the negate operation of line 33), otherwise, it stands positive, and the method ends.

The COS: method in line 38 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 67), 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 Yerk definitions are added (lines 69-70). 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 72 - 75
Class Angle provides an example of how the sin and cos definitions in lines 69 and 70 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 Yerk 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 Integer 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 Yerk word, Sin. With Yerk.com 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.

```
76      (  Turtle Graphics Objects for Demo )
77
78      Decimal
79      ( Define a turtle-graphics pen)
80      :CLASS Pen  <Super Object
81              ( 1st 5 Ivars comprise a PenState structure)
82              Point   PnLoc          \ location of pen
83              Point   PnSize                 \ width,  height
84              Int     PnMode
85              Var     PnPatLo
86              Var     PnPatHi
87              Angle   Direction
88              Point   homeLoc
89              Int     maxReps
90              Int     initLen
91              Int     deltaLen               \ change in len
92              Int     deltaDeg               \ change in angle
93
94              :M  GET:    (ABS)  call GetPenSt  ;M       \ save state here
95              :M  SET:    (ABS)  call SetPenSt  ;M       \  restore from here
96
97              ( deg -- )
98              :M  TURN:  +: Direction  ;M
99
100             :M  NORTH:   0 Put: Direction  ;M
101
102             ( x y -- ) ( Draw a line to x,y if pen shows)
103             :M  MOVETO:  Set: Self  Pack  call LineTo  Get: Self  ;M
104
105             ( d  -- ) ( Draw d bits in current direction)
106             :M  MOVE:   {  Dist -- }
107                     set: self  Cos: Direction  dist * 10000 /
108                     Sin: Direction dist  * 10000 / negate
109                     Pack call Line  get: self    ;M
```

```
110
111             ( x y -- ) ( Goto a location without drawing)
112             :M  GOTO:    Put:  PnLoc     ;M
```

```
113
114             ( x y -- ) ( set the center coordinates)
115             :M  CENTER:  put: homeLoc  ;M
116
117             ( -- ) ( Place Pen in center of Forth Window)
118             :M  HOME:   get: homeLoc  Goto: Self   ;M
119
120             ( w h -- ) ( Set size in pixels of drawing pen)
121             :M  SIZE:      Put: PnSize     ;M
122             ( x y w h mode  -- )
123             :M  INIT:   Put:  PnMode  Put: PnSize   Put: PnLoc    ;M
124
125             ( initlen dLen dDeg -- ) ( set parameters)
126             :M  PUTRANGE:  put: deltaDeg  put: deltaLen  put: initLen  ;M
127
128             ( maxReps -- )
129             :M  PUTMAX:  put: maxReps  ;M
130
131             :M  CLASSINIT:   Get: self  200 put: maxReps ;M
132
133             ( Draw a spiral of line segments - Logo POLYSPI)
134             :M  SPIRAL:  { \ dist  degrees delta  reps -- }
135                     home: self
136                     get: initLen -> dist  get: deltaLen -> delta
137                     get: deltaDeg -> degrees 0 -> reps
138                     BEGIN  1 ++> reps reps get: maxReps <
139                     WHILE
140                             dist Move: Self  degrees  Turn: Self
141                             delta ++> dist
142                     REPEAT  ;M
143
144             ( n -- ) ( Dragon curves from Martin Gardner)
145             :M  DRAGON:  Dup  0=
146                     IF  Get: deltaLen  Move: Self  Drop
147                     ELSE  Dup  0 >
148                             IF  Dup 1- Dragon: Self
149                                     Get: DeltaDeg Turn: Self
150                                     1 swap - Dragon: Self
151                             ELSE  -1 over - Dragon: Self
152                                     360 Get: deltaDeg - turn: Self
153                                     1+ Dragon: Self
154                             THEN
155                     THEN  ;M
156
157             \ draw an infinite Lissajous figure
```

```
158             :M  LJ:  { \ reps -- }
159             up: self  0 -> reps
160             get: initLen get: direction * cos 120 /  getX: homeLoc +
161             get: deltalen get: direction * sin 120 /  negate getY: homeLoc +
```

```
162             goto: self
163             BEGIN   1 ++> reps  reps  get: maxReps  <
164             WHILE
165                     get: initLen get: direction * cos 120 / getX: homeLoc +
166                     get: deltaLen get: direction * sin 120 / negate
167                     getY: homeLoc + moveTo: self
168                     get: deltaDeg turn: self
169             REPEAT  ;M
170     ;CLASS
171
172     \ Define a Smalltalk Polygon object as subclass of Pen
173     :CLASS Poly  <Super Pen
174             Int     Sides   \ # of sides in the Polygon
175             Int     Length  \ of each side
176
177             :M  DRAW: { \ turnAngle -- }
178                     360 Get: Sides  / -> turnAngle
179                     Get: Sides 0
180                     DO   Get:  Length  Move: Self
181                             turnAngle  Turn: Self
182                     LOOP ;M
183
184             ( len  #sides -- ) ( Store sides and go to Home)
185             :M  SIZE:  Get: Self  Put: Sides  Put: Length
186                     Home:  Self   up: Self  ;M
187
188             \ Spin a series of polygons around a point
189             :M  SPIN: { \ reps -- } Home: self  10 Get: InitLen Size: self
190                     0 -> reps
191                     BEGIN  reps  get: maxReps  <
192                     WHILE   Draw:  Self  Get: deltaDeg  Turn: Self
193                     Get: deltaLen +: Length  1 ++> reps
194             REPEAT  ;M
195
196             \ Default Poly is 30-dot triangle
197             :M  CLASSINIT:  30 3 Size: self  100 put: maxReps  ;M
198
199     ;CLASS
200
201     \ Create a pen named Bic
202     Pen Bic
203
204     \ Create a Polygon name Anna
205     Poly Anna
206     60 4  Size: Anna
```

Line 78

The program begins with a declaration that all numbers to follow will be in decimal. 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 79 - 92
Beginning on line 105 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 Yerk'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 Yerk on the Mac. As you can see in lines 82 - 92, 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 earlier in the Class Angle (lines 72 - 75).

As the comment in line 81 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.

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 94 - 95
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 97 - 131
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 100) 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 103) features a Yerk word that's new to you: Pack. First of all, the stack notation (line 102) 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:

```
0->255 20  <RETURN>
2->.s      <RETURN>
Parameter Stack:
              20  $      14
             225  $      FF
2->pack     <RETURN>
1->.s       <RETURN>
Parameter Stack:
          310975  $    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 Pack operation 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 105 - 109 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 Yerk before Turtle.

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 108. 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. The Mac screen is 512 pixels horizontally by 342 vertically. A full window, like yerk.com, is roughly 500 x 320, give or take a few pixels.

INIT: (line 123) 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 126'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 131, 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 133 - 170
In these lines are three methods that are largely Yerk 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 134, SPIRAL: declares four local variable names: dist, degrees, delta and reps. In line 135, the pen is moved to the center of the current drawing window. Dist and delta are given values in line 136 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, degree, gets its value in line 137 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 138 - 142. 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 Yerk, 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 170.

Lines 172 - 199
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 201 - 206
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 206, 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 Yerk words, defining new subclasses and even modifying the existing

methods to do some tricks. The more you play with Yerk, the quicker you will become comfortable with all its powers.

First, you must load the file struct1 and sin (only if you have turned off the computer since the last lesson) and the Turtle source files in that order. Load each file 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 Yerk 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:

```
   ( n1 n2 n3 --  )
: lj  cls  putrange: Bic
                250 160 center: Bic
                lj: bic   cr        ;
```

Try typing in various three integer combinations (e.g., 9 11 301 lj <RETURN>) 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 Yerk 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:

```
   ( n1 n2 n3 --  )
: 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 message to Bic to change the PUTMAX: parameter:

```
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 Yerk prompt really messed things up. Therefore, define a Yerk 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 yerk.com.

If you have come to this lesson without turning off your Mac or quitting yerk.com from the last lesson, then you should remove all of Turtle's code from yerk.com. The fastest way to do this is to use Yerk'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 <RETURN>

and then select List Words from the Utilities pull-down 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 yerk.com and load struct1 and Sin. Our Logo program will load atop Sin.

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 Yerk 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:

```
    (  --  HiPat  LoPat  mode w h x y )
:M  INSPECT:  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:

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

Here is the listing of Yerk 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

( Erase old Logo command onscreen and reposition prompt )
: SPOT  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 ;

( dist --  )
```

```
: FORWARD   -curs   undraw: turtle   move: turtle
        draw: turtle   .ok  ;

( deg --  )
: LEFT    turn  ;

( deg -- )
: RIGHT  negate  turn  ;

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

The above Yerk 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 Yerk 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 the Editor, save the modified source as "Logo" (perform a Save As... operation from the Editor menu). Close the Editor and return to the Yerk.COM window. Select Echo During Load from the Yerk menu. Then load Logo into memory with the Load selection from the File menu or by typing:

0->// logo <RETURN>

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. As noted in the chapter on the Editor, various other messages, like "object not unique" and "method redefined," will scroll by on the screen. As long as the load doesn't stop, however, nothing fatal is occurring in memory. When the load is complete, clear the screen (CLS <RETURN>) 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 Yerk, 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 YERK-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 Yerk glossary and class-object-message relationships.

In the remaining lessons of this tutorial, we'll be exploring some of YERK'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 the 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" and "dmenu.txt" source files, which are located in the Demo folder (within the supplement folder) on the Yerk disk. 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 file: Window, ctl, ctlWind, VScroll, and QD.

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

to load grDemo into yerk.com, first make sure that the following source files are on your working Yerk disk in the internal drive:

ctl
VScroll
ctlWind
Sin
Turtle
grDemo
demo.load

Next, double-click yerk.com from the desktop to load yerk.com into memory. Then select Load...from the File menu. When the dialog box appears, select demo.load and open it. This file contains a list of load commands for each of the class files listed above. As each file loads, you'll see a series of messages about various words and methods being redefined. When all the files are loaded (the File title on the menubar reverts to black on white), type:

dstart <RETURN>

After a bit of disk activity, the program, Yerk 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 Yerk. 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. From running the

demonstration program you will see that this program is a demonstration of the Mac's ability to control parameters by way of controls, like scroll bars.

Secondly, it demonstrates the Mac's ability to produce 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 lessons. And third, this program shows how pull down menus control the actions of a program.

If you were designing this program, this would be the time when you ask yourself what kind of objects will be in the program so you can establish what classes need to be defined. Only two classes are defined in this program: one for the controls and one for the demonstration window that holds those controls.

The first of two class definitions is called VSCtl, and it defines the characteristics of a special kind of vertical scroll bar that also displays a digital readout of the control's thumb setting in a little box below the control.

Macintosh 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 Control Panel of the Desk Accessories. By adjusting the knob with the mouse, you immediately adjust the volume of the internal beeper. 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.

A scroll bar is another kind of control. It's the same kind of scroll bar you're familiar with from MacWrite. 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.

An important concept to know about controls is that they must be "owned" by a window. That is to say, one of the specifications for a control is the window in which the control is to be located.

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 Yerk 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 Controls

The special scroll bar controls in grDemo inherit their instance variables from the superclasses VScroll and Control. The list of available ivars includes an integer for the definition ID, an Xarray for the addresses of a scroll bar's five possible actions, and an Ordered-col for the actions' corresponding part numbers. In the subclass VSCtl, two ivars are added: a rectangle that specifies the location of the digital readout box for each control and another, slightly smaller rectangle inside the readout rectangle where the digital figures appear.

As soon as a VSCtl object is created, the CLASSINIT: method of its superclass (VScroll) 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.

In the DISPLAY: method, the "cursor" where the digits are to be placed is positioned one pixel up from the bottom left corner of the viewreadout rectangle. Whatever numbers were there previously are cleared, and the new digits (retrieved by get: super) are printed in a field of 3 digits.

DRAW: is an extension of DRAW: in Class Rect and DISPLAY: in this class. It will be used to redraw the readout box and the digits whenever the program window needs to be updated (e.g., when it is dragged partially off and then back on the screen).

The PUT: method sets the control's value in the control superclass (precisely the opposite of the GET: super message above in the DISPLAY: method). This method also calls the preceding SHOW: method, which displays the updated control and its readout rectangle value. This seemingly tiny put: method is actually doing a lot of work each time a control is adjusted.

The NEW: method, with the aid of named input parameters, builds a new control and establishes the location of its readout box. The parameters it needs are the coordinates of the top left corner, the vertical length of the control, and the address of the owning window. From the top left coordinate and length values, the NEW: method in the Control superclass calculates the bottom right corner coordinates.

Three text attribute statements set the digits' 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, the fonts are assigned ID numbers. They are as follows:

SystemFont (Chicago)	=	0
ApplicationFont (Geneva)	=	1
New York	=	2
Geneva	=	3
Monaco	=	4
Venice	=	5
London	=	6
Athens	=	7
San Francisco	=	8
Toronto	=	9

While in this list the application font is the same as Geneva, in some programs, a special applications font is inserted in its place (the Seattle font in Multiplan, for example). For the digits in the readout box in grDemo, then, the Geneva font was selected.

Next, the coordinates of the readout rectangle are calculated from the named input parameters. You should be able to follow the Yerk math here, but in case you can't, the location of the readout box is derived from the location of the scroll bar. By taking various coordinate points from the control's rectangle, it is possible to define the readout rectangle as a box 4 bits wider on each side, starting four bits below the bottom of the scroll bar, and extending 20 bits below the bottom of the scroll bar. The coordinates are then stored (put:) in the control's rectangle ivar, readout. The rectangle is drawn by the draw: readout message. Those same coordinates are fetched (get: readout) and fed to the viewReadOut. From there, viewReadOut's coordinates are inset three pixels on a side.

While the first three methods of class VSCtl may be invoked repeatedly during program execution, the NEW: method will be summoned only once for each scroll bar. But this is the method that fills the scroll bar objects' ivars with enough values for them to become truly functional scroll bars.

Following the VSCtl class definition are three lines of code that create the objects that will become the scroll bars. They are given the names VS1, VS2, and VS3, respectively. They have no real "life" yet, because they need to receive a NEW: selector in a message. But these object-creation statements make dictionary entries for these three objects.

Declaring Some Constants

Five values are created next. The first pair are the coordinate point on the full 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 figures will be recalled later when it comes time to create the window for the program.

The fifth value, vsLen, is the length of any scroll bar that will go inside that window. Notice that the number placed in this value is derived from the top and bottom values of the program window. The value was calculated in this way so that if we decide later to change the dimensions of the program window (by changing the values of the window coordinates), the length of the scroll bars will be adjusted accordingly. We won't have to hunt through the source code for all references to the scroll bar length to effect the change throughout the program.

End of lesson 18

Lesson 19

Windows

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 1-19.

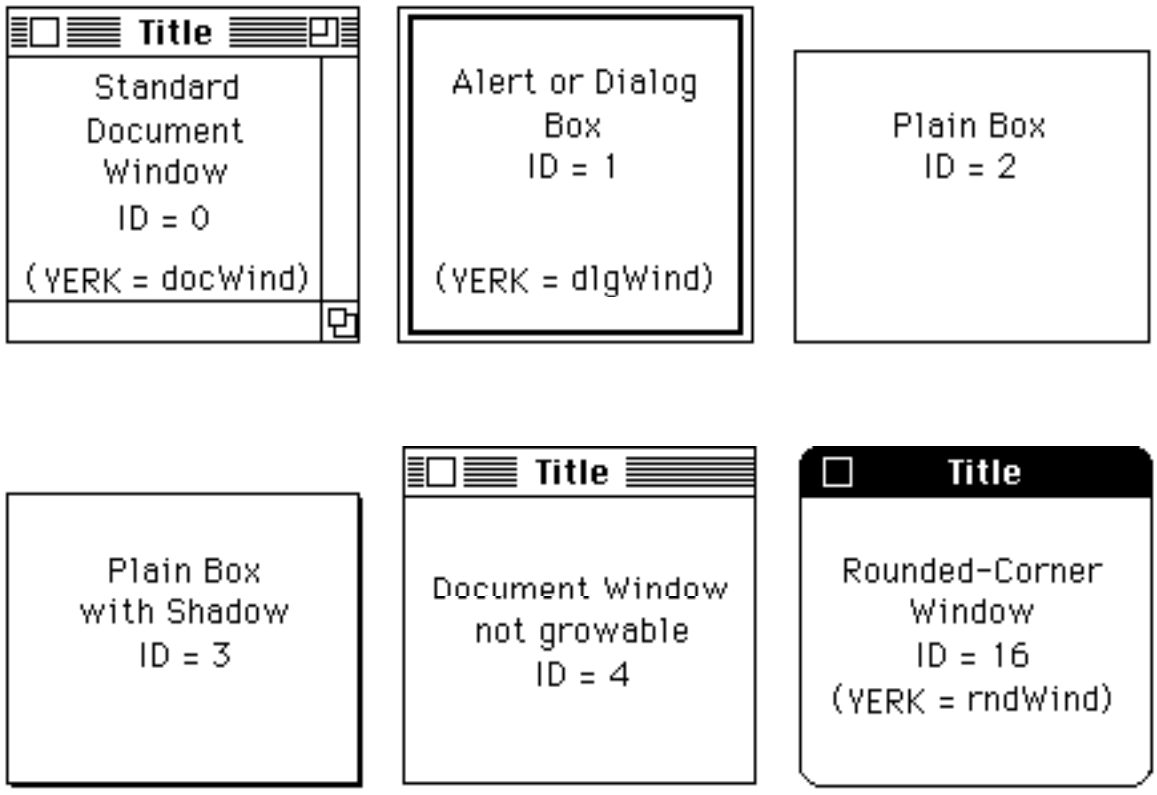


Figure 1-19

Whenever you define a new window, choose one of the window types by number. Yerk 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 Yerk constant names are shown in Figure 1-19. This Yerk demo graphics program also uses rndWind, but any window style could have been selected.

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. 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 GrDemo Window

Class grWind, a subclass of CtlWind (itself a subclass of Window) lays the framework for the window of this graphics demonstration. One additional ivar (over and above a standard window's ivars and CtlWind's ivars) is the rectangle where the graphics will be drawn.

Let's jump to the CLASSINIT: method first, since this is executed the instant we create the window. First of all, it summons the CLASSINIT: of the superclass, Class Window. This puts most of the ivars in order for us. The remainder of the CLASSINIT: method defines the coordinate points of the graphics window (thePane).

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. Fortunately, 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), and flags for being visible on the screen and having no close box -- are supplied within the method or as constants already defined. Once all the factors are safely on the stack in the proper order, the method calls the superclass' NEW: method. The NEW: method also establishes the area within which the grWind object will be draggable. We wanted this to be done at run time, since 'grayRgn' calculates the largest possible region your Mac has (due to different and multiple screens). If we had wanted to limit the drag region to a fixed area, we could have defined this at compile time. If you have the program running now, try dragging the window.

DRAW: is an important method, and one you should remember when you write programs that have windows that need updating. 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. That's what DRAW: manages. The object that receives a draw: message is set to be the current window. All three digital rectangles of the scroll bars are redrawn. Between the BeginUpd and EndUpd Toolbox calls are messages that redraw the scroll bars, confine the update region to the drawing rectangle, and execute the word stored in the object's draw ivar. Clip: contrect restores the clip region to the entire program window. Details about updates and clip regions can be found in the Quickdraw chapter of Inside Macintosh.

While this demo does not demonstrate it, another, faster way of updating a window is to draw to an

offscreen bitmap. There is an optional class called 'copier' that will allow you to do this. If you would like to see how this works (after you play with the grdemo program a bit) redefine dstart to not enter in the Begin Again loop. That way, after you execute dstart, the program will now start and the fwind and interpreter will be available for interactive testing. Load the source 'offscreen' located in folder 'My stuff'. Then type in the following lines:

```
copier bob
dwind destport: bob
4 15 320 220 destrect: bob
new: bob set: fwind
```

The messages to send to the object 'bob' are save: and draw:....try it to see how it works. You might want to reset the destport to fwind, or just use the example in the 'offscreen' source. There is an additional source called 'copywind' that creates a copier object to save the contents of a window instance's content rect. Try the examples here...

The Demo Window

Returning to the grDemo, next comes the statement that creates the window, dwind, which is an object of the class grWind we just defined. In the next line of code the title of the dwind window is made a string constant labeled DTITLE.

The phrase Set: fWind causes graphics output to be sent to the fWind grafPort during compilation, because it executes directly. This is so we can keep track of what is occurring during the load.

A new definition, @dParms, fetches the current readings of each control. This definition is a shortcut that allows us to use one word to do the work of three messages for each of the four following definitions.

The words of these four 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.

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.

NewObjs defines, in one word, an important and powerful series of operations that will take place at the beginning of program execution. First, it closes the main yerk.com window (fwind). Then it opens dwind with the title assigned to dTitle. And then it passes all necessary parameters to activate each of the scroll bars. These parameters were detailed in the new: method of class VSctl, above.

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 three string constants, AB1, AB2, and AB3. Following that come three program lines that define what is to happen when that selection is made. It selects the system font (Chicago) in 12-point, positions the cursor at point 8,40, and "types" the three strings on the screen.

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 rectangle that defines the bounds of the window. That's because if you were to drag the window to the lower left corner of the

screen, the pen and polygon must still find the center of the graphics rectangle. By measuring the coordinates with respect to the window -- and not the entire Macintosh screen -- the Mac has no problem finding the precise spot, no matter where on the screen you drag the window.

Scroll Bar Actions

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 Yerk 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 is coupled to a toolkit call that automatically calculates a value based on the relative position of the thumb along the range of the scroll bar. After that, it updates the window. 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 'c ("tick c") returns the address (specifically, the cfa) of the word that follow it. In this case, the cfa 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 cfa of a graphics routine (lj, spin, etc.) in the draw ivar of dwind. This will all come together at the end of the program.

Next, the cfas of the five control actions are stored in each scroll bar's actions ivars. The syntax here, 5 'cfas ("five tick cfas") is a shortcut for entering 'cs for each action handler word. Addresses for each definition are passed as parameters to the scroll bars' actions ivars.

Menus

Older versions of Yerk created menus from an external text file which you created with an Editor. However, the current version no longer supports this method; instead menus are defined in a resource file.

Menus work in ways 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.

GrDemo's menu text is created by a resource editor such as Apple's ResEdit. The menus reside in a resource file called demo.rsrc and have an ID number associated with them. By convention, the Apple menu is ID=1. We've assigned ID=128 to GrafMen. These id's match the assignments in the grdemo source. The Apple menu is loaded with its handlers in the

grdemo source at compile time. Only two need be loaded; the desk accessories will be loaded automatically because of the way class Applemenu is defined. Notice that the program instantiates only Grafmen, since Applemen is part of the Yerk itself.

The first selection will execute the word 'about'. The second selection is the dividing line between About Curves and the balance of the items in the menu which does nothing, so a null is the appropriate handler. This menu item is not meant to be active.

Selections for the Grafmen menu are rather straightforward. "Graphics" is the heading that will appear on the menu bar. Then there are four menu handler words that we'll define in grDemo in a moment. To Quit the program, the menu handler word is sayonara which will be defined in the demo as the plain ol' Yerk word, bye, which returns you to the desktop.

Next come the menu handler word definitions for Grafmen. Each one checks the appropriate menu item making use of the globals mitem and theMenu...these are set automatically when you select a menu item. You could have hard-coded the item numbers and GrafMen instead, but this method illustrates the convenience and power of using a late bound object (theMenu). Each handler then places the cfa of the drawing word in the draw ivar of dwind and 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.

The word that brings the menus to life is 'getGRMenu'. It lists the menus you want to load, the total number of the menus, and then sends the init: message to the latebound menubar object. The resource file must have been opened prior to the call.

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, dstart. The odd-looking coordinates (1000,20) are indeed out of the range of the Mac's visible screen. Newobjs, as defined earlier, brings the dwind and VS objects to life. Bic and Anna are centered in the graphics rectangle, their respective maximum repetitions are set, ranges are set for the first pen action, the first figure is drawn (update:), and the cursor is turned off. At the end, the BEGIN...AGAIN loop effectively disables the keyboard, since any key entry is immediately dropped from the stack. The program essentially loops here indefinitely, yet it is always "listening" to mouse events as they affect controls and menus.

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

In Summary

Now that you have seen the entire grDemo program, you should notice some key points about Yerk 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.

If you write a program that you want to "seal off" as a self-running program, see Chapter 5, of Part II, for details on how to Install a Yerk application.

Where To Go From Here

You've already had quite an exposure to Yerk and object oriented programming. You've seen how Yerk 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 Yerk programming on the Macintosh.

It is important that you have an acquaintance with the powers of the predefined classes and the words in the Yerk 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 Yerk 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 on the disks. 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 Yerk the faster you'll be comfortable with it.