

5. Loops

Overview

Human beings get very bored quickly when they are forced to repeat a task over and over again. Human beings get very bored quickly when they are forced to repeat a task over and over again. (See what we mean?)

This is one of the reasons why computers are so useful -- they're very good at carrying out repetitive actions very quickly without complaining. The process of getting the computer to repeat the same task, over and over again (until we tell it to stop), is known as *looping* or *iteration*. In this chapter, we'll be examining two ways to get the computer to repeat a task using the *loop* construct and the *repeat* construct.

The Loop Construct

A loop repeats a series of instructions for a user-defined number of cycles or *iterations*. The number of times that the loop repeats itself is usually predetermined and kept track of by a *loop counter*. The logic for a loop is shown in Figure 5.1. First, we set the start value of the loop counter. For this example, we'll set it to 1. The loop counter is initialized to this value. Then we set the desired end value of the loop counter, in this case, 10. This determines how many times we wish to execute the loop (number of iterations = counter end value - counter start value + 1, or a total of 10 times in this example). Within the loop, we execute the instructions we wish to repeat (the loop contents) once. Following this, the value of the loop counter is increased by the step size (in this case, 1) to indicate that the loop has been executed once. The current value of the counter is compared to the desired counter end value. If the current value of the counter is less than the desired end value, the loop contents are executed again and the loop counter checked again, etc. Once the loop counter value is equal to that of the end value, we exit the loop.

```
short  counter,  
       start = 1,  
       end = 10,  
       step = 1;
```

```
for (counter = start; counter <= end;  
    counter = counter + step) {  
    // contents of loop  
}
```

Figure 5.1: Logic for a loop as written in the C++ language

In Prograph, a *loop* construct repeats a task until a *match* test within the repeated code signals that a given condition required to end the loop has been met. That is, the loop repeats until a *fail*, *terminate* or *finish* control has been activated.

Let's look at how loops work in Prograph. We'll start by creating a simple counting loop. Create a new Counter project, section and universal method. Now create a second universal method called Count For Me and complete its code diagram as shown in Figure 5.2. The +1 primitive (with no space between the "+" and the "1") increments the number input to it.

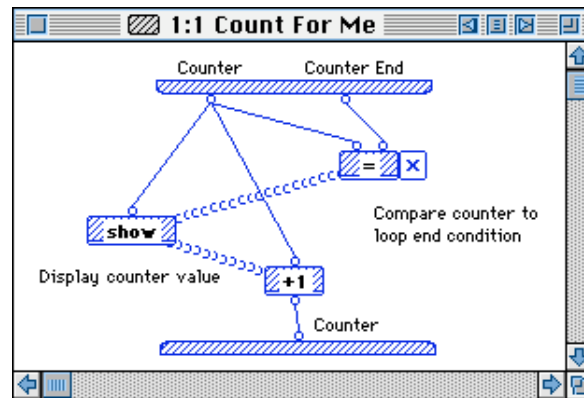


Figure 5.2: Initial code for the Count For Me method

The Count For Me method contains the *loop contents* -- the actions to be carried out during each pass through the loop. In this example program, the loop contents will simply increment the value of the loop counter and display its value with a *show* primitive. It also contains a match test for whether or not the necessary conditions to end the loop have been met; that is, if the value of the loop counter is equal to that of the counter end value, which holds the number of times we want the loop to execute.

By default, the = primitive has an attached control with an X in it. Remember that this X in the control means “go to the next case immediately if this test *fails*.” In other words, this case will be stopped when the test indicates that the value of the loop counter is *not* equal to that of the counter end value. This is the opposite of what we really want. We want the loop to *stop* when the value of the loop counter is *equal to* counter end -- that is, when the = primitive’s match test *succeeds*. We must therefore change the X in the control to a √. Select the option to do so from the Controls Menu.

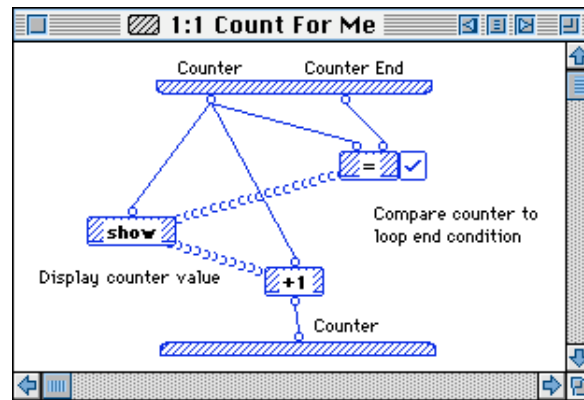


Figure 5.3: Partially-corrected code for the Count For Me method

There’s still one problem remaining. Notice that the control attached to the = primitive lacks bars on its top and bottom symbolizing method input and output bars. This plain control is a Next Case control. This states that the loop contents method should *continue* looping after the match test comparing the value of the loop counter and the counter end value succeeds. In other words, the loop will go on executing. We want to *prevent* the loop from continuing once this loop’s contents have executed for this iteration. We use the Finish option in the Controls Menu to change this control to a Finish control. Now, if the = primitive’s test succeeds, we’ll *finish* executing this iteration of this method (Count For Me), then *stop looping*.

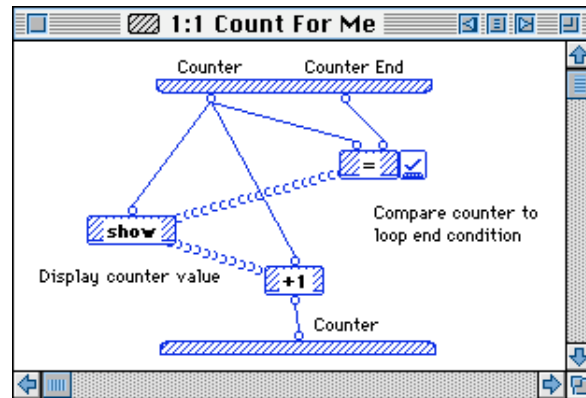


Figure 5.4: Completed code for the Count For Me method

Let's review what the Count For Me method -- the loop contents -- does. When it is first called, a starting value and an ending value for the loop counter are input to the method as variables. The value of the loop counter is tested to see if it's equal to the counter end value. If not, the value of the loop counter is displayed, then incremented and output from the method. Upon the *next* iteration of the loop, the process is repeated, but now the *new* (incremented) value of the loop counter will be again tested, displayed and incremented. If the = match test is successful, the code within this method would finish executing, but no further looping would be allowed to occur.

Now let's write the Counter method, which will begin the looping process. Enter the following code diagram. It asks the user for a starting and ending value for the loop counter, and feeds these numbers into the Count For Me method. The Count For Me method has two input nodes for the loop counter and the counter end value, and one output node for the incremented value of the loop counter.

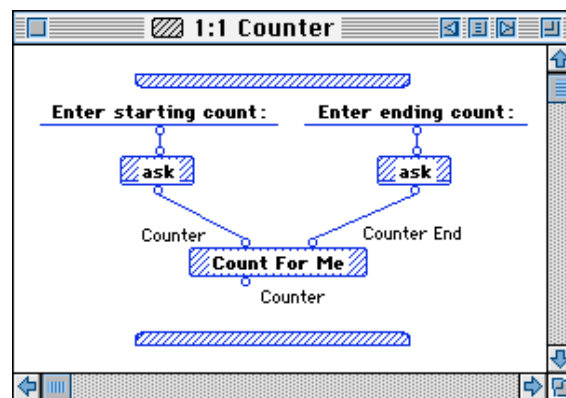
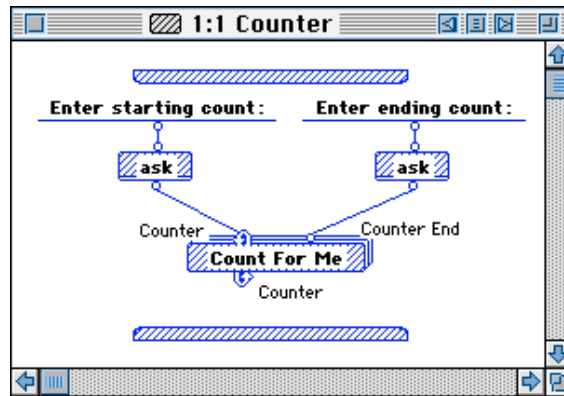


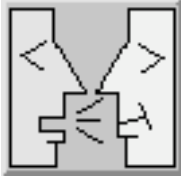
Figure 5.5: Initial code for the Counter method

So far the Count For Me method is doing nothing out of the ordinary. What is telling it to perform a loop at all? At present, *nothing*. We must specifically *tell* the Count For Me method to loop itself. Highlight the leftmost root node of the method icon, the one that feeds the value of counter to Count For Me. Hold down the shift key and also highlight the terminal node of the method that outputs the incremented value of counter. Now select the Loop option from the Controls Menu. The window should now look like Figure 5.6.

**Figure 5.6: Calling the Count For Me method in a loop**

The arrows forming a continuous looping path from the input node to the output node of Count For Me signify that Count For Me now is being looped. They also denote that the output of the method -- the incremented value of the loop counter -- is *fed back into the input* of Count For Me when it executes the next iteration of the loop. Now Count For Me will be called over and over again and the value of the loop counter incremented again and again until the = test within Count For Me finally succeeds (when the loop counter equals the counter end value). On each iteration of the loop, the current value of the loop counter will be displayed by the Count For Me method's show primitive.

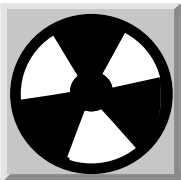
The stacked loop symbol with its arrows helps us distinguish at a glance that this is indeed a loop. In addition, just as the use of multiple case windows helped make matches understandable, keeping the contents of a loop in its own individual window makes it easy to note what part of your code is within the loop.



By The Way...

Every iteration through this loop will display the value of the loop counter in a dialog box, which you'll have to dismiss by hitting OK or the Return key. Unless you like doing this over and over, we suggest that you use values of start and end that are close to each other to restrict the number of times the program will loop.

Note that in this example program, we have a match logical test within a loop construct. In Prograph, it's still simple to tell where one code execution control construct ends and the next begins via their differently shaped icons and independent code windows. In textual languages like C++, your sole clue to the end of one construct and the beginning of another is how well you've indented and commented your source code text.



Warning!

Every so often, when loops are not constructed properly, or when some action within a loop contains an error, the loop can go haywire. You can wind up with what's called an "*endless*" or "*infinite*" loop -- one that goes on forever without stopping! If this should happen, hold down the Command key and press the period key, then hit Return. Usually, this will abort execution of the loop, and you can then select Step/Show Off and Stop Running from the Execution Menu to continue editing the program.

Exercise 5.1:

Write a program called Square Roots Table which outputs the numbers from 50 to 40 (backwards), along with their square roots.

Prograph Version of a For...Next Loop

Let's do a variation on the loop. In many computer languages such as BASIC and C, there are built-in looping commands called For...Next loops. This construct allows you to create loops where the counter is increased by *any* step size (not just by 1), like 2 or 3, etc., or even decreased by any number upon each cycle of the loop. We'll now create the Prograph equivalent of a For...Next loop that counts

backwards. Our For...Next loop can be reused in future programs that might need flexible loops.

Create a new Count Backwards program and section, containing three universal methods called Count Backwards, Check Values and For-Next Loop. The Check Values method will make sure that the starting value for the loop's counter is higher than the value of the end value of the counter. If so, these values are passed through unchanged (see Figure 5.7). If not, their values are swapped to ensure that the counter starting value is now the larger of the two, and a message is displayed informing the user that the swap was made (Figure 5.8).

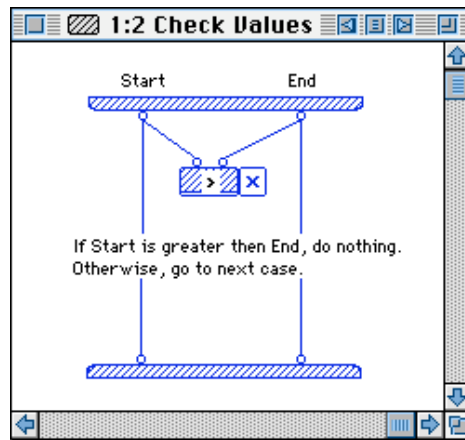


Figure 5.7: Case 1 of the Check Values method

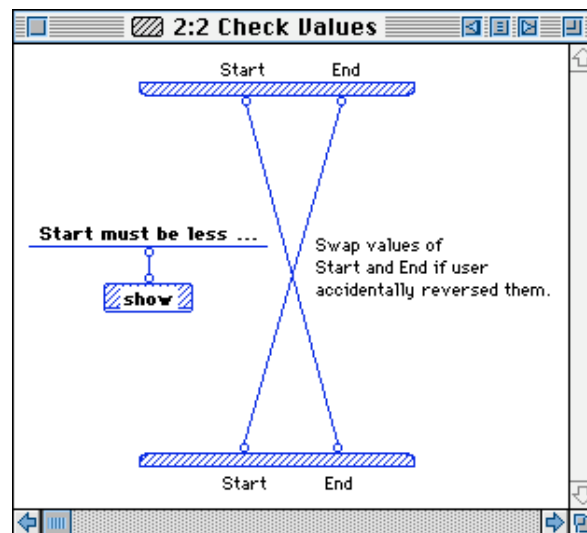


Figure 5.8: Case 2 of the Check Values method

The For-Next Loop method code diagram -- the contents of the loop -- is shown in Figure 5.9. The value of start, end and step (the amount that the counter changes on each iteration through the loop) are inputs to the method. The value of start is decremented by the value indicated by step using the “-” (negation) primitive. The new value of start is then checked to see if it’s less than that of end. If not, the value of start is output and the loop continues to execute. If so, the For-Next Loop method finishes but cannot be used in the loop again, so the loop ends.

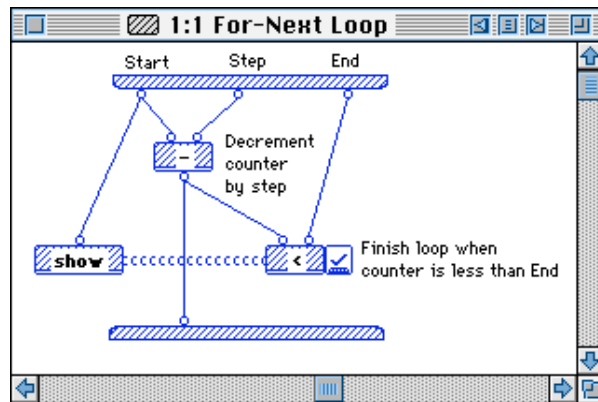


Figure 5.9: The For-Next Loop method

All that remains is to write the Count Backwards method that gets the loop going. Remember, to convert the For-Next Loop method into a looped method, highlight the For-Next Loop method icon’s leftmost root node (the start value) and its terminal node, then select the Loop item from the Controls Menu. The loop will only affect these two nodes, so it is only the value of start that will change with each pass through the loop. The values of end and step always remain the same value that the user enters for them.

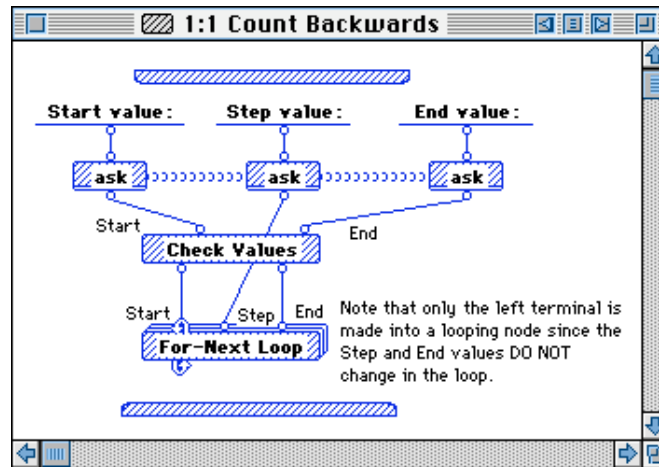


Figure 5.10: The Count Backwards method

Exercise 5.2:

Write a program called Multiplication Table to ask for number, then display in successive iterations the number times the value of the loop counter (starting with the counter equal to 0). For example, if the user inputs a 4, the program will respond with:

(Iteration 1) 0 times 4 equals 0.

(Iteration 2) 1 times 4 equals 4.

(Iteration 3) 2 times 4 equals 8.

...

(Iteration 11) 10 times 4 equals 40.

Looping an Indefinite Number of Times

Although loops typically are made to iterate a fixed number of times, there is nothing that really forces them to be used this way. Our final example of a loop will demonstrate how to construct a loop that can execute any number of times depending upon the task required. We will construct a program to calculate the square root of a number. Actually, Prograph contains the `sqrt` primitive to do this for us. We thought that we'd show you what programmers had to do when such routines were not written for them by language developers. We'll use one particular algorithm for finding square roots from scratch, using a "trial-and-error" looping technique. To understand how the program will work, see Figure 5.11.

Number to be square-rooted: 14

What is done...

	<u>Root 1</u>	<u>Root 2</u>
Start with 1 and divide the original number	1	14
Average roots, divide this result into the original	7.5	1.867
Continue averaging & dividing into original	4.863	2.879
Note: Average=Root 1, Quotient=Root 2	3.871	3.617
Keep doing this until Root 1 is close to Root 2	3.744	3.739
The programmer decides how close Root 1 is to Root 2 -- CLOSE ENOUGH HERE!	3.7415	3.7418

Output the Square Root of 14: 3.742

Figure 5.11: Algorithm for the Square Root program

The computer can repeat the loop -- averaging, dividing, and testing whether the two roots are nearly equal -- many times very quickly. We can arbitrarily choose an extremely small number such as 0.000001 to be the difference between the two roots required for the loop to stop. Just how many times the loop executes is determined by how many times the averaging, dividing and testing must be carried out before the small difference between the two roots is reached. There's no way for us to know this in advance since it will differ for each number we wish to calculate a square root.

Create a new program, section and universal method called Square Root. The Square Root method, shown in Figure 5.12, will simply call two other methods, Get Number and Output Root, in turn.

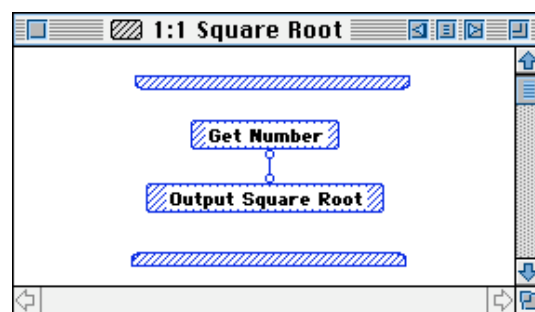


Figure 5.12: The Square Root method

Create and open the Get Number method by double-clicking on its method icon in the Square Root code window. This method (see Figure 5.13) will ask the user to enter a number, then will pass the number through the abs primitive to ensure that the number is

positive (just in case the user accidentally enters a negative number) since the square root of a negative number doesn't yield a real number.

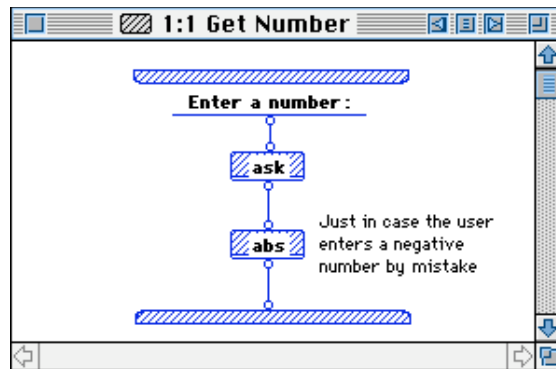


Figure 5.13: The Get Number method

Create a new method called Calculate in the Universal window. The Calculate method will serve as the loop contents -- the method that will keep averaging and dividing until its two roots are "equal" (actually *within 0.00001* of each other). Complete this method as shown in Figure 5.14.

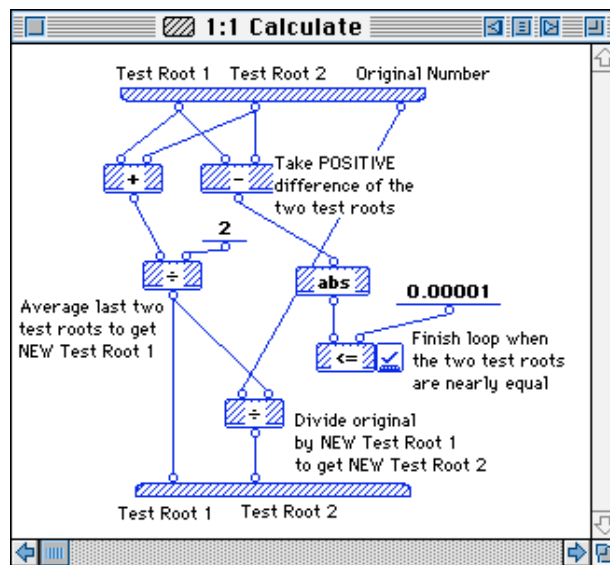


Figure 5.14: The Calculate method (loop contents)

The Output Root method calls the Calculate method in a loop, then prints the final output of the Calculate method when the loop ends. To create the loop, change only the Test Root 1 and Test Root 2 nodes of the method into looping variables. The value of Original

Number will not be changed. Also note the link between Test Root 2's loop symbol and the show primitive. Even though Test Root 2's output node on the Calculate method icon is involved in a loop, it still is an output node and can be used as such! When the loop is exited, the final output value of Test Root 2 is sent from this terminal node to the show primitive for display to the user.

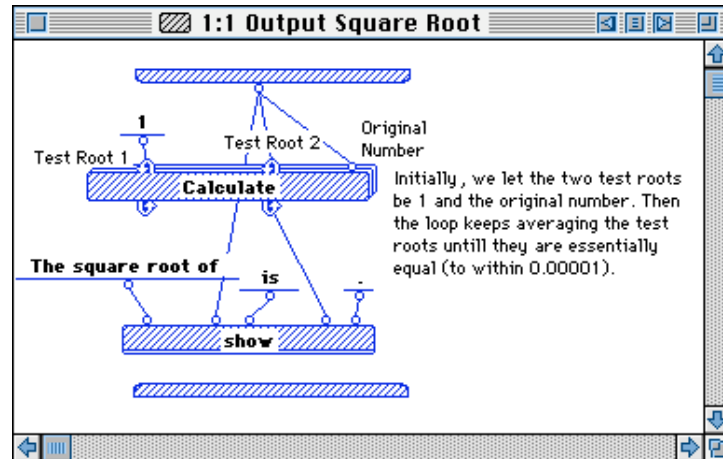


Figure 5.15: The Output Square Root method

Exercise 5.3:

Write a program called Division which simulates the integer division and modulus functions *without using the \div (or idiv) primitives!* The program should ask for a dividend and divisor, then show the quotient and remainder.

The Repeat Construct

The *repeat*, like the *loop*, carries out a task over and over again until a *terminate* or *finish* control has been activated. However, the *repeat* and *loop* are used differently. While a *loop* requires that some data value be passed as a *counter* or some other indicator of when to end the loop into and out of the method that's looping each time it's executed, the *repeat* doesn't have this requirement.

The *repeat* causes a method to be *repeated indefinitely until some particular condition is met*. For example, a method could be repeated until the user responds "No" to the prompt "Do you want to try again?". This process is very similar to the Pascal language's REPEAT...UNTIL command or C++'s do...while command. We'll

construct a program that will use a “Do you want to try again?” prompt and a *repeat* construct to perform a repetitive action. In this program, the repetitive action will be to present multiplication problems to the user (using random numbers from 0 to 15 as multipliers) and prompting the user to enter answers until they are correct. Then the user will be asked if they wish to try again, and if so, the whole process will repeat.

Create a new program, section and universal method named Math Quiz, then complete its code window as shown. The Instructions method simply displays the message “You’ll be asked to multiply two numbers. Keep trying until you get it right.”. The Quiz method is where all of the action really takes place. To make the Quiz method one which will repeat, highlight its method icon, then select the Repeat item from the Controls menu. Notice that its icon changes to one that looks like a stacked set of methods (see Figure 5.16). This stacked icon immediately tells you that the method will repeat over and over again just like a loop’s stacked icon did.

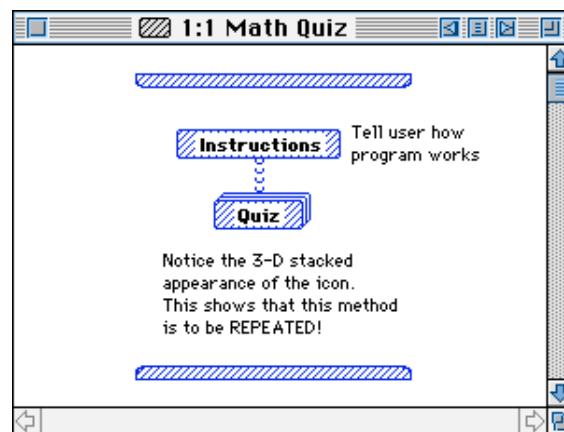


Figure 5.16: The Math Quiz method with a repeated Quiz method

Now let’s write the contents of the repeating code contained in the Quiz method (Figure 5.17). This method will get Generate Numbers to make the random numbers that will be multiplied, then calls Ask & Check to present the question to the user and see if they’re correct. Finally, it asks the user if they wish to do all of this again. If the user answers “yes”, the *repeat* is carried out again. If not, the Quiz method is exited.

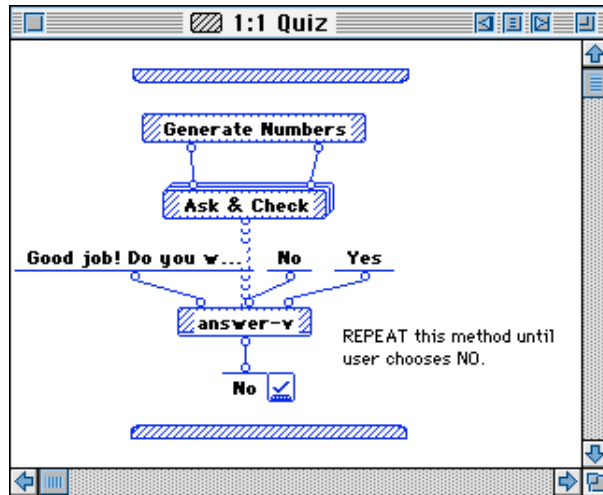


Figure 5.17: The Quiz method with a nested looping method

The Ask & Check method code is shown in Figure 5.18. It is a second repeat within the repeating Quiz method. This “loop-within-a-loop” structure is known as a *nested loop*. The *outer* repeat, Quiz, keeps initiating new questions until the user wants to quit. The *inner* repeat, Ask & Check, asks the multiplication question and keeps checking the user’s answer to see if it’s correct. If not, it continues asking the same question until a correct answer is given.

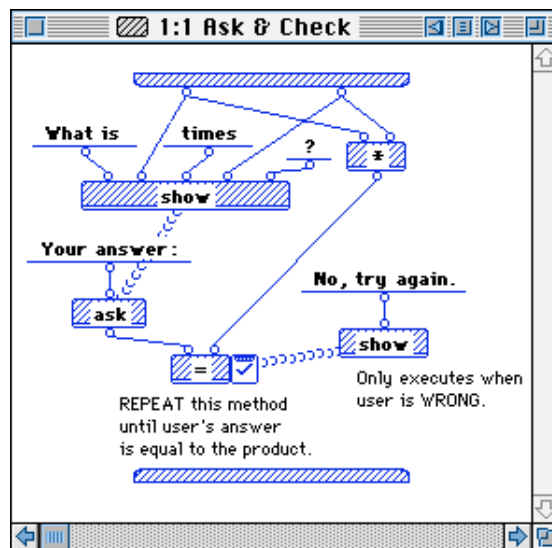


Figure 5.18: The Ask & Check method

The Generate Numbers method (Figure 5.19) simply generates two random numbers between 0 and 15, which will be used as the multipliers in the multiplication problems presented to the user.

The `rand` primitive generates an integer number between 0 and $2^{31} - 1$. This number is an integer, which we'll divide by 15 with the `÷÷` integer division primitive to get a quotient (unused here) and a remainder between 0 and 15 -- our random number. Note that the second output node for the remainder on the `÷÷` primitive is not present by default. It is an optional node that we have to add to the primitive icon.

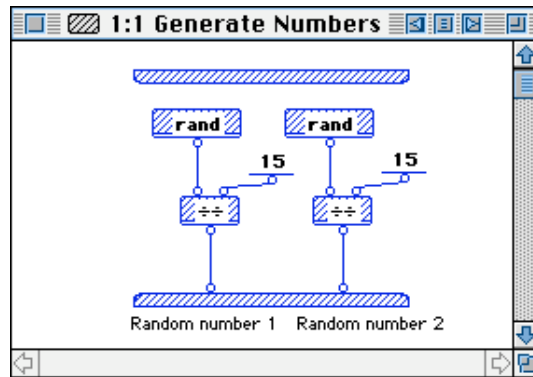


Figure 5.19: The Generate Numbers method

Exercise 5.4:

Modify the Math Quiz program to make a new program called Square Root Quiz. The program should instruct the user to enter a number whose square root will be calculated. The program should keep asking for numbers and displaying their square roots until a negative number is entered. At that point, the message "Cannot square root a negative number!" should be displayed, and the program ended.

Summary

In this chapter, we introduced two Prograph constructs for executing repetitive blocks of code:

- The *loop* construct is used to execute a block of code a specific number of times. It usually serves as the Prograph equivalent of a low-level *for-next* loop. We have provided code examples that extend the utility of the *loop* so that it provides *all* of the general functions of a *for-next* loop.

- The ***repeat*** construct simply executes a block of code over and over again until a specific condition is met. Its equivalent in other programming languages is the *while* or *do-while* loop.
- Loops, repeats and matches can be nested within each other to form more complex constructs. Unlike in textual source code, in Prograph it's simple to see where one construct ends and the next begins.