

4. Program Flow Control

Overview

Just as calculations are the heart of any computer program, *decision-making* commands are just as essential for writing large and flexible programs. Programs need to be able to handle choices made by the user, make decisions about how to handle data and deal with different data values produced at different times by your methods or external code written in other languages such as operating system routines. Prograph provides a number of decision-making program elements to allow you to control the flow of your program according to the situation at hand.

In this chapter, we'll examine Prograph's decision-making constructs -- matches, controls and injects -- and how they regulate the flow of a program by directing execution to different portions of code. In addition, we'll discuss using recursion in Prograph programs.

Decision-making and Cases

Decisions are carried out by creating more than one *case* of code for a method. Every method contains by default a *single* case, shown when you first open a method's code window. If no decision-making commands are used in the method, this single case is sufficient on its own. But once a decision has to be made, the program requires more than one case -- more than one possible set of instructions that can be carried out. If the decision has one particular outcome, one case's code will be executed. If another outcome results, a different case's code must be executed instead. This logic is shown in Figure 4.1 in C++ language code. We'll show you how this works below.

```

    if (condition1) {
        // case 2
    }
    else {
        // case 1
    }

```

Figure 4.1: Decision-making and cases

The Match Construct

Matches are essentially “*true-false*” tests, similar to the “if-then” and “if-then-else” constructs of the C++ language. The match checks if a given condition is met or not, and executes one or another *case* or block of code accordingly. Let's put a match to work. Starting with a fresh project and section, create a universal method named Toss A Coin and open its default case window. The default window of a method is case 1 out of 1 possible cases, hence the “*1:1*” (case 1 out of 1) in front of the method name in the

window title. We will write a method to ask the user for the result of a coin toss (“heads” or “tails”). On the basis of the coin toss, we’ll display an appropriate message.

Write the code shown in Figure 4.2 for the Toss A Coin method. The `answer` primitive will prompt the user with a message to select one of the buttons presented in a dialog box. The names of the buttons, as well as the prompt, are the inputs to the `answer` primitive. Its output is the name of the button clicked by the user. We’ll use the prompt “Heads or tails?”, and name the buttons “Heads” and “Tails,” respectively.

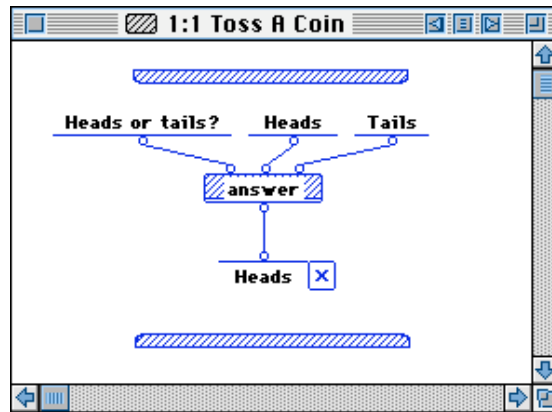


Figure 4.2: User selection and logical decision-making processes in the Toss A Coin method

The match below the `answer` primitive call will check to see if the user selected the “Heads” button in the dialog box by reading the value returned by the `answer` primitive and checking for a match to the text “Heads”. What will the match do after it's checked for the value of this string? This depends upon the type of *control* in the box on the right-hand side of the match icon. The “X” symbol in the control indicates that the match will direct the flow of the code so that it *goes to the next case if the match fails*. That is, if the string returned by `answer` is *not* “Heads”, the code contained in a second case window (which we haven't created yet) will be executed. If the match *succeeds*, the remaining code in case window 1 (the one we've been working in) will be executed instead. This is equivalent to the C++ code of Figure 4.3.

```
#include <streams.hp>
#include <string.h>

void
TossACoin( void )
{
    char toss[ 32 ];

    cout << "Select heads or tails: ";
    cin >> toss;

    if ( ! strcmp( toss, heads ) ) {
```

```

        // case 2
    }
    else {
        // case 1
    }
}

```

Figure 4.3: Equivalent C++ code for the decision-making process of the Toss A Coin method

Now we must provide the code that the method will execute depending upon the button selected by the user. We now know that the match will go to another case window if the match test fails. Our task for the case when the match succeeds is simpler -- the remaining code in *the existing* case window will be executed. So let's put some code there to be executed. We'll just have the computer display the message "You chose Heads!" with a `show` primitive.

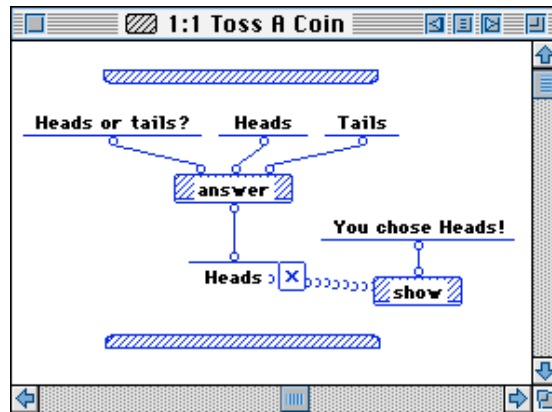


Figure 4.4: First case of the decision-making process

We still only have one case at present, shown in this default code window (window 1:1). We now need a second case for the code to be executed when the match *fails*. How do we create one? Notice the three symbols in the right-hand side of the title bar of the Toss A Coin window, shown in Figure 4.5.



Figure 4.5: Case control buttons in the case window title bar

These are case control buttons, used to list and display all of the cases of a method. Move the mouse to the *middle* control -- the Case List button -- and select it.

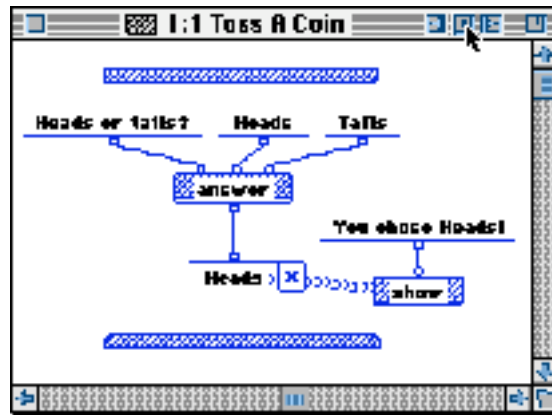


Figure 4.6: Selecting the Case List button

At the bottom of the window, a list called a case palette (see Figure 4.7) appears showing all of the cases that have already been created. At present, only one case exists -- shown in the default Toss A Coin method window -- so the list only contains a symbol with a 1 in it.

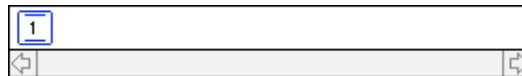


Figure 4.7: Case Palette at the bottom of a case window

Create a second item in the case palette next to the 1 symbol. This shows that we've now created case 2, (see Figure 4.8). The case indicator numbers in the title of the Toss A Coin window now reflect the fact that there are 2 *case windows*, with the currently visible window now containing case 1 out of 2 possible cases (1:2).

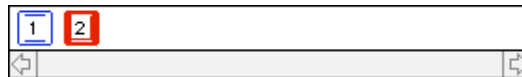


Figure 4.8: Adding a second case to a method with the Case Palette

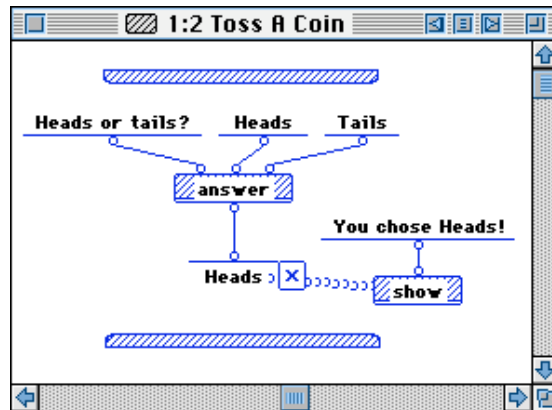
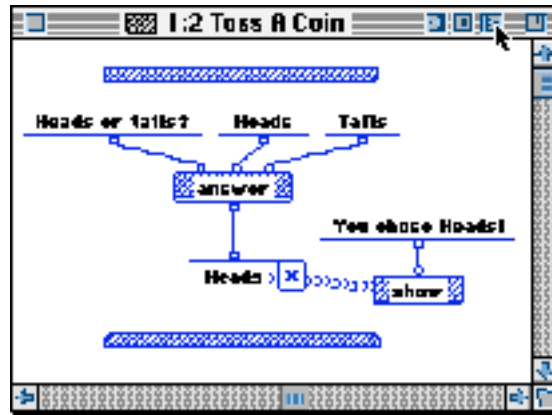
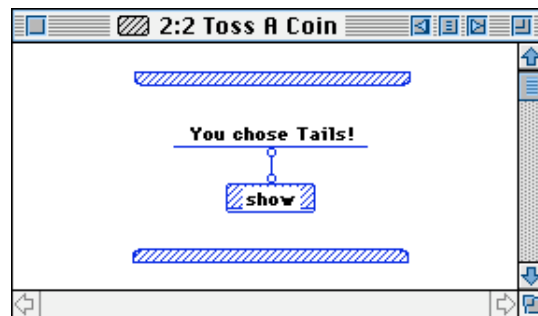


Figure 4.9: First case window after creating a second case

Open the window for case 2 by selecting the Next Case button in the window title bar (Figure 4.10). Notice that the title of this window indicates that this is case 2 out of 2 possible cases (2:2).

**Figure 4.10: Moving to the next case window**

In the new window, create a show primitive to display the text “You chose tails!”.

**Figure 4.11: The completed second case of the Toss A Coin method**

The Toss A Coin program is now complete. Let’s go through the code, step by step. When the Toss A Coin method starts, the user is presented with a dialog box that asks them to select whether a coin toss is “heads” or “tails” by selecting one of two buttons in the dialog. The response (either “Heads” or “Tails”) is sent to a match, which tests whether the text is equal to “Heads.” If it is not (the test failed), case 2’s code is executed, and a message is displayed stating that “Tails” was selected. If the text indeed *was* “Heads”, case 1 continues and a message is displayed stating that “Heads” was selected.

Having two separate windows for the two cases visually reinforces the two possible paths of execution of the Toss A Coin method, where one given case will

execute depending upon the outcome of a logical *match* operator. This immediate recognition of individual cases will be even more evident for *large* blocks of code. In textual languages, as the blocks of code after a logical statement get larger, it gets harder and harder to tell where one block ends and the next block begins, making it difficult to trace the alternative paths of execution. In Prograph, you just can't make that kind of mistake reading your code.

Exercise 4.1:
Write a method called Ask Opinion that asks the user if they enjoy using Prograph, and gives them a choice of responding Yes or No. If they choose Yes, respond with "I'm glad you like it!". If they choose No, respond with "Sorry...but Prograph grows on you!".

Multiple Cases

Your programs can now carry out different actions according to the selections the user makes. But what if the decision to be made is not a simple yes-no (or true-false) decision? For example, what if we wanted to test if a student received either a passing grade on an exam or a failing grade, or if we just accidentally typed in an invalid grade? This is not a simple true-false decision since there are *three* categories of grades -- passing, failing or invalid. Fortunately, matches may be *combined* to produce more complex decision-making rules, much like the *switch* statement in C++, whose logic is shown in Figure 4.12.

```

switch (selector) {
  case condition 1:
    // execute case 1 code
    break;
  case condition 2:
    // execute case 2 code
    break;
  case condition 3:
    // execute case 3 code
    break;
}

```

Figure 4.12: The logic of a multiple case decision

Create a new project with a section and method each named Grade Exams. Open this method's case window and enter the code diagram pictured in Figure 4.13.

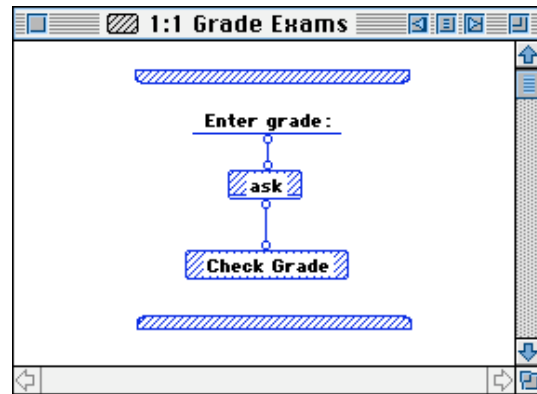


Figure 4.13: The Grade Exams method

Now open (and create) the Check Grade method's case window. Notice that the input bar will contain an input node since this method expects one input number: the grade. Complete the case window as shown in Figure 4.14. The \geq (greater than or equal) and \leq (less than or equal) primitive names may also be typed in as ">=" and "<=".

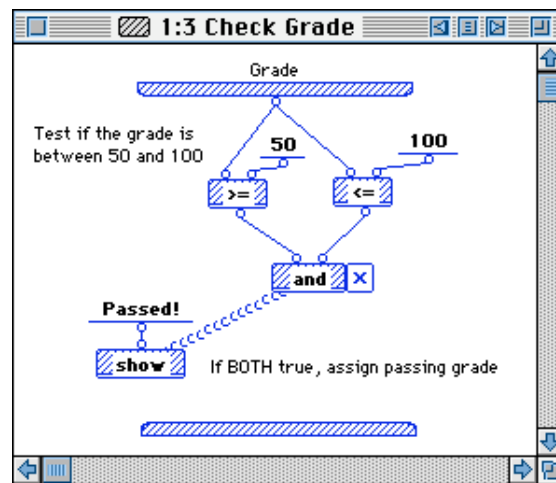


Figure 4.14: First case of the Check Grade method

In this first *case* of Check Grade, the grade that is fed into the method is subjected to two logical tests -- to see if it is greater than or equal to 50, our cut-off for a passing grade, and if it's less than or equal to 100, the highest possible grade. *Both* of these tests must succeed if the remaining code in this case window is to be executed. We ensure that *both* tests' outcomes are combined into *one larger logical test* for the *range* of grades between 50 and 100 by using the logical and primitive, which itself performs a match test, as can be seen by the match *control* on its primitive icon. The outcomes of *both* the \geq test *and* the \leq test must be *successful* if the and test is to succeed. If *either* the \geq or \leq tests *fail*, the and test will *fail*.

If the and test succeeds, that is, if the grade is between 50 and 100, case 1 displays the message "Passed!". If *any one* of the \geq , \leq or and tests fails, the *second case*

is entered. Let's define the second case as shown in Figure 5.15. Remember that you need to click on the Case List button in the window's title bar to open the case palette and create new cases.

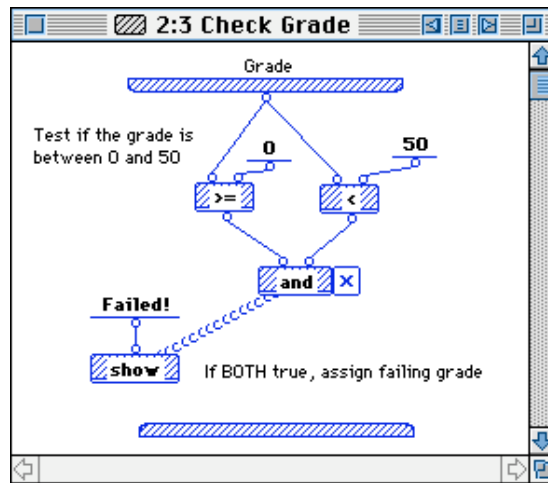


Figure 4.15: Second case of the Check Grade method

In the *second case*, the grade is tested once again with the aid of the and primitive. If the grade is between 0 and 50, this case's tests are successful and the remaining code of case 2 is executed -- the message "Failed!" is displayed. If any of case 2's tests fail, a *third case* is executed.

Case 3 is much simpler. The method will only reach the code in case 3 if the tests in *both* case 1 and case 2 fail. That is, case 3 is executed only if the grade entered is *invalid* -- less than 0 or greater than 100. The message "Invalid grade!" is displayed in this event.

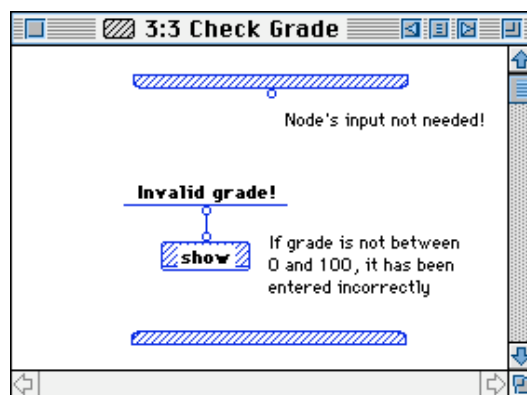


Figure 4.16: Third case of the Check Grade method

Exercise 4.2:

Write a program called Play Dice that plays a dice game. Generate a random number from 2 to 12 to simulate the tossing of two dice. If the roll is a 2, output “Snake eyes! You lose!”. If the roll is a 12, output “Box cars! You lose!”. If the roll is a 7 or 11, output “A natural! You win!”. For any other roll, simply output “Your point is ”, followed by that number.

Hint: To generate a random number from A to B, inclusive, first integer-divide (the \div primitive) a random integer number obtained with the `rand` primitive by the range of numbers desired (B-A+1). For example, if you wish to obtain random numbers from 10 to 20 (A=10, B=20), divide the output of the `rand` primitive by (B-A+1) or (20-10+1), that is, 11. Then add the lower bound of the range (A) to the *remainder* of this division. In the case of our example, that would require adding A=10 to the remainder 9.

Let’s review one more example of using matches and multiple cases, since the concept of having many code windows for one method can be difficult to grasp at first. Create a new project, section and method called Choose An Action. Complete the Choose An Action method’s first case window as shown in Figure 4.17.

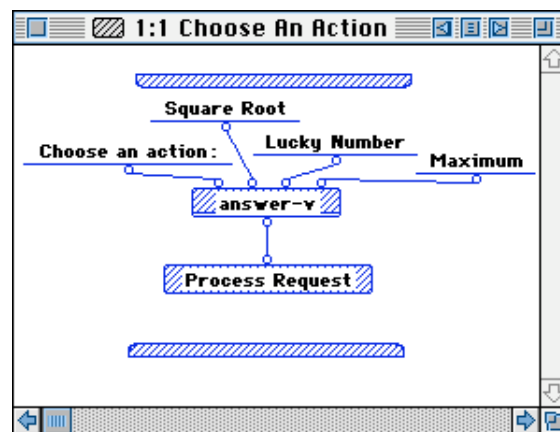


Figure 4.17: The Choose An Action method

We use the `answer-v` primitive to present a vertical list of three possible actions to be selected by the user: calculating the square root of a number, choosing a (random) “lucky number,” and finding the maximum of two numbers. The dialog box that prompts the user for a selection is shown below. Once the choice has been made, it is sent to a universal method called Process Request.

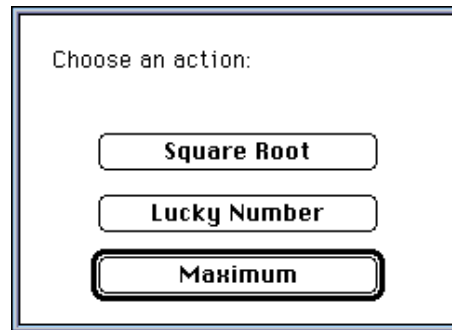


Figure 4.18: Dialog box presented by the `answer-v` primitive

Open the first (default) case window of Process Request. Complete the case window's code diagram as follows in Figure 4.19. This case will check if the selection made was "Square Root". If not, the next case will be entered. In this first case, we use the `sqrt` primitive, which calculates the square root of a number. The `show` primitive then displays the square root to the user.

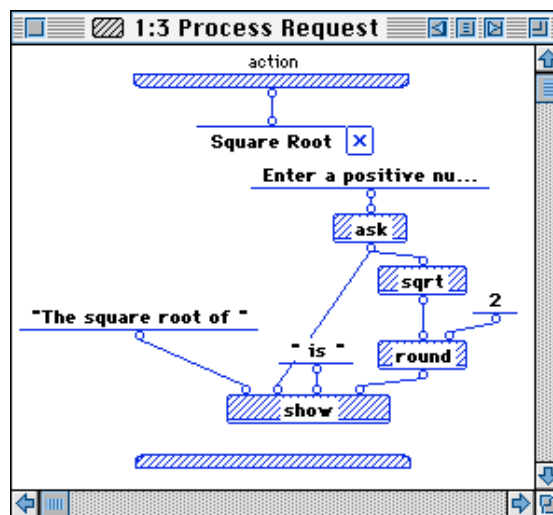


Figure 4.19: First case of the Process Request method

Open the *case palette* by clicking on the *Case List* button and create two additional cases for this method. The second case will test if the user selection was "Lucky Number". If it was not selected, the third case will be entered. If "Lucky Number" was selected, we use the `rand` primitive to generate a random positive number to be displayed by a `show` primitive.

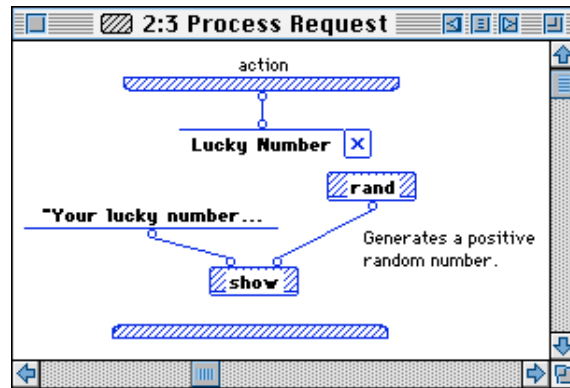


Figure 4.20: Second case of the Process Request method

The final case is entered only if the first two cases' matches fail. This method compares two numbers input by the user with the `max` primitive and displays the maximum of the two numbers.

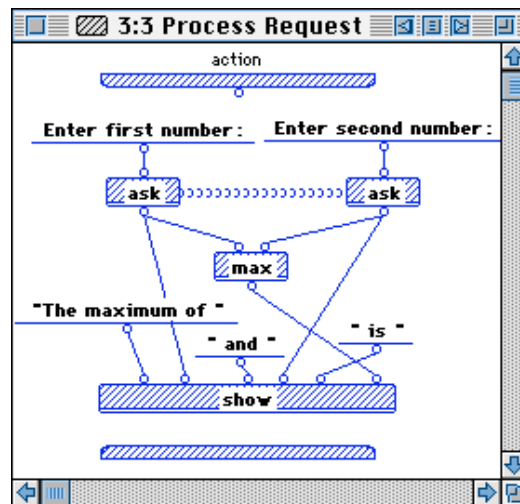


Figure 4.21: Third case of the Process Request method

This example program has shown a simple way to allow the user to select what actions a program should take by presenting a dialog box of possible actions and applying a match test to route the code execution to the appropriate case. By the simple combination of the `answer-v` primitive to prompt the user for a choice and a `match` construct, we have allowed the program to take several courses of action at run-time.

Exercise 4.3:

Create a Prograph method called Trig Functions that asks the user to enter an angle (in degrees), then asks the user to select whether to calculate the angle's sine, cosine or tangent. Depending upon the user's choice, the program should calculate the sine, cosine or tangent and display the appropriate result. Use the `round` primitive to round off the results to five decimal places before displaying it.

The Control Construct

We've seen how the *match* construct allows a program to modify its actions according to user selections or data values. However, up to this point, we've only used one type of *control* on a match -- testing for *failure* of the match test. Matches are actually more versatile than this, and may perform other types of actions according to the *control* we choose to associate with the match.

We've also seen that matches are not the only way to make decisions in a program. Decision-making may also be done by *logical primitives* such as the *and* and *or* primitives, which themselves contain match tests.

Next Case on Failure

Just to review, the *next case on failure* control's symbol is an X, which means "go to the *next case immediately if the match test fails*." In other words, if the match *fails*, the current case's code is stopped *right at the point of the match test*, and the program execution jumps to the next case's code. In the example below, the next case is entered if the number tested *fails* to be equal to zero. If the test *succeeds* (the number *is* equal to zero), the rest of the code in the current case is executed.



Figure 4.22: Failure control on a match -- Go to next case on failure

Next Case on Success

The *next case on success* control's symbol is a \checkmark , which means "go to the *next case immediately if the match test succeeds*." In the example below, the next case is entered if the number tested *is* equal to zero. If the test *fails* (the number is *not* equal to zero), the rest of the code in the current case is executed.



Figure 4.23: Success control on a match -- Go to next case on success

When match icons are first created, they are by default tests for *failure*. To change the control of a match so that it enters the next case on *success* of the test, use the Failure-Success Conversion item in the Controls Menu.

Continue



Figure 4.24: The Continue control

The *continue* control allows the remainder of the code in the current case to *continue* executing even if the match test fails. We use only the “continue on failure” notation since “continue on success” would be an unnecessary test.

The *continue* control has two bars in its icon -- one at its top and one at its bottom -- to represent the method's input bar and output bar. This indicates that the *continue* control allows the method to continue its flow from the method's start to its end.

Terminate



Figure 4.25: The Terminate control

If the match conditions are met, the *terminate* control ensures that the current case is *exited immediately* and its remaining code is *not* executed. For example, in Figure 4.25, if the number the match receives as an input is not equal to zero, the current case is exited and a second case is entered. This is especially useful for repetitive methods like *loops* and *repeats*, which we'll discuss in the next chapter. Not only does *terminate* stop the current execution of the method, it also prevents further *repetition* of the method.

The *terminate* control has one bar in its icon at its top to represent the method's input bar. The output bar is not represented, indicating that the *terminate* control prevents the method from completing its flow to the method's end.

Finish



Figure 4.26: The Finish control

The *finish* control is similar to *terminate*. However, if the match conditions are met, the *finish* control allows the remaining code of the current case to be executed once, but it prevents further repetition of the method in *loops* and *repeats*.

The *finish* control has one bar in its icon at its bottom to represent the method's output bar. There is no input bar picture, indicating that the *finish* control allows the method to complete its flow to its end (but *not* start *again* in a repetition).

Inject

Methods such as primitives that are called by your program have predefined names. You must know before you can run your program exactly which methods you'll call and what those methods' precise names are. But what if you *don't* know in advance which method to run? What if you need the user of the program to make that choice at run-time, so the interpreter won't know which method to call until the program is run?

The *inject* construct is a tool for calling a method whose name is defined only at run-time. Let's see just how useful an *inject* can be. We'll write a program that will ask the user for an angle, then ask what calculation to do on this angle. According to the user's selection, the appropriate calculation will be done.

Create a new program, section and method called Choose Trig Function, and fill the Choose Trig Function case window as shown in Figure 4.27.

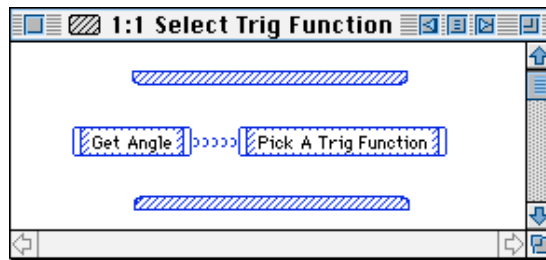


Figure 4.27: The Select Trig Function method

Remember that trigonometric primitives such as `sin`, `cos` and `tan` require that the input angle be expressed in radians rather than degrees. Earlier in the book, you were asked to write two methods to convert angles from degrees to radians and from radians to degrees. Those methods should have been saved in a section called Angle Conversions. We're going to make use of the methods in that section again by using the Add Section item of the File Menu. This program will be the first that you'll write containing more than one section. Add Section allows you to load those parts of another program that you wish to reuse and merge them with a program you're working on now. After adding the Angle Conversions to the current program, you'll see that the Sections window now contains the new section. The Universal Methods window of that section contains the methods Degrees To Radians and Radians To Degrees that you saved previously. We can now use these two methods in our program.

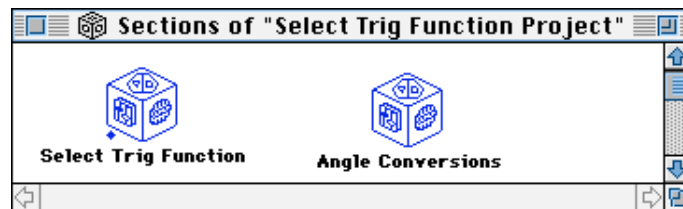


Figure 4.28: Angle Conversions section added to the current program

Complete the code diagram for the Get Angle local method as shown in Figure 4.29. We use the Degrees To Radians conversion method to be sure that angle returned by the Get Angle method is in the proper format for the calculations we'll do on the angle later. The Get Angle method will output the angle both in degrees (to be displayed at the end of the program) and in radians (for calculations).

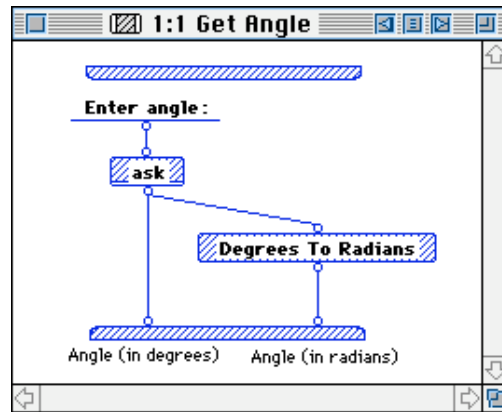


Figure 4.29: The Get Angles local method

The Pick a Trig Function method is simpler. It uses the `answer-v` primitive to ask the user to pick a trigonometric calculation to perform on the angle. Notice the names that will be placed in the buttons of `answer-v`'s dialog box. They are "*sin*", "*cos*" and "*tan*", the names of the primitives used to perform the sine, cosine and tangent calculations. The name of the trigonometric primitive that will be called next is selected by the user, returned by `answer-v`, and output by the Pick a Trig Function method.

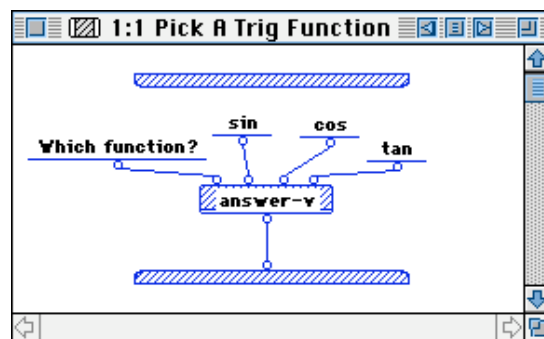


Figure 4.30: The Pick A Trig Function local method

Now we must make an operation that will call the requested trigonometric primitive. We could add match tests and three cases that would call either the `sin`, `cos` or `tan` primitives. Or we could use the *inject* operation. Return to the Select Trig Function case window and create a new program element below the Pick a Trig Function icon. Give it two input terminal nodes. The left node will receive the angle from the Get Angle local method. The right node receives the output of Pick a Trig Function.

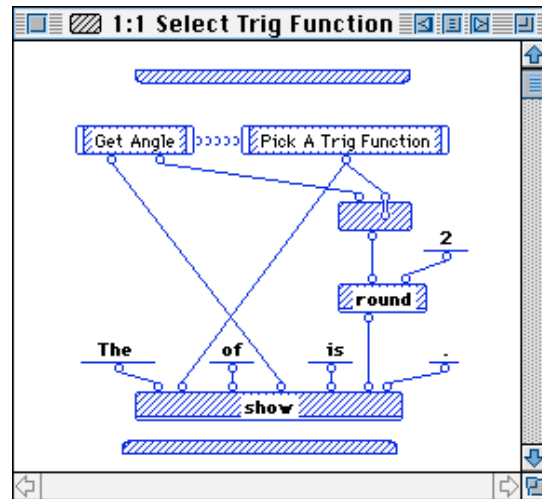
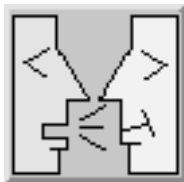


Figure 4.31: The Select Trig Function method

The right root node of this operation is an *inject* node, and the blank operation looks like a *nameless* method. The shape of the *inject* node actually looks as if it's injecting something into the method icon, and it is -- *this method's name*. The method will take the place of a match and three cases with calls to the *sin*, *cos*, or *tan* primitives. The *inject* node tells this method to call the primitive whose *name* is output by the *Pick a Trig Function* local method.



By The Way...

The inject node is also used widely throughout the *Application Builder Classes* library code that will form the basis for Prograph program user interfaces. Injects are used to determine what code to call when a particular user interface element is selected by the user of the program.

Since all three of the possible primitives that may be called require one input -- the angle -- and one output -- the result of their calculation -- we must include nodes for the three primitives' input and output on this method icon. That is why we have two input terminals and one output root on this injected method -- one input and one output to serve as the trigonometric primitives' input and output, and one input terminal to inject the name of which trigonometric primitive to call.

Now our program will take the angle entered by the user and calculate its *sine* if the user selects "*sin*", its *cosine* if "*cos*" is selected, or its *tangent* if "*tan*" is selected. The last step is to display the answer with a *show* primitive. However, trigonometric calculations will produce *real* numbers with *many* decimal places. Do we really want to display all of those decimal places? Probably not. We'll *truncate* the number, that is, limit the number of decimal places displayed to two places with a primitive called *round*.

Exercise 4.4:

Write a program called Triangles that offers the user the following menu selections:

- Find Hypotenuse
- Find Leg
- Find Angles

Each choice by the user will call the appropriately named method (each of which you will write). Find Hypotenuse will ask the user for two legs of a triangle and output the hypotenuse of the triangle. Find Leg should ask the user for one leg and the hypotenuse, then output the second leg. Find Angles will ask for the opposite leg and the hypotenuse, then display the size of the angles of the triangle.

Recursion

As a final lesson in structured programming, we'll examine an example of the use of *recursion* -- the calling of a method by its self. In mathematics, an exclamation point written after a number means *factorial*. For example, $7!$ means "*seven factorial*". Factorial notation is used to calculate statistical probability (permutations and combinations). It's a short way to represent a multiplication of a number by all integers smaller than itself. For example,

$$6! = 6 * 5 * 4 * 3 * 2 * 1$$

$$7! = 7 * 6 * 5 * 4 * 3 * 2 * 1$$

By definition, $1!$ and $0!$ both are equal to 1.

Note that $n! = n * (n-1)!$. For example, $7! = 7 * 6!$, and $6! = 6 * 5!$, etc. This technique of defining one factorial multiplication in terms of another simpler multiplication is what *recursion* is all about. In this lesson, we'll use recursion to calculate the factorial of a number.

Create methods called Recursion and Factorial. Open the Recursion case window and complete its code diagram as shown in Figure 4.32.

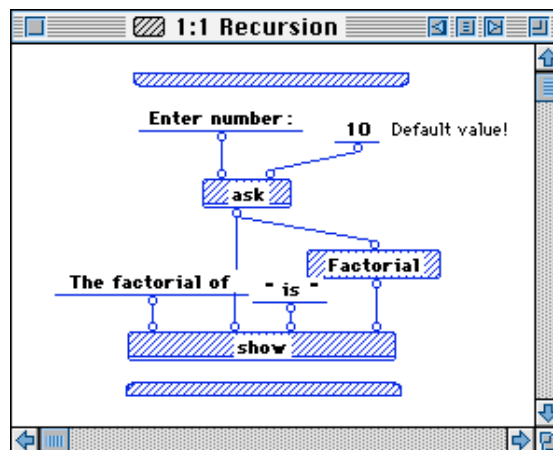
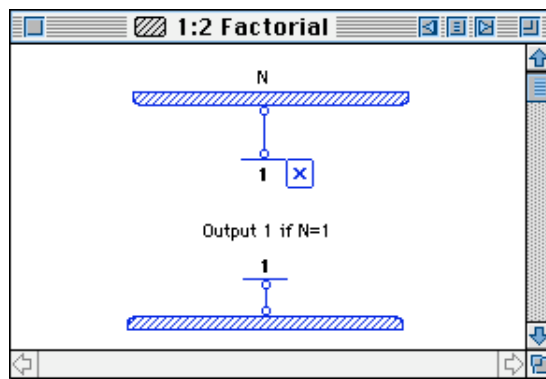


Figure 4.32: The Recursion method

Notice the right node on the **ask** primitive. This second node makes it possible to provide a *default value* that will appear each time the **ask** dialog is presented. In this program, the prompt “Enter number to factorial:” is presented, along with the number 10 in the text-entry portion of the dialog. If the user wants to use the value 10, all he or she has to do is hit the Return key and this *default* value of 10 will be used by the program.

Now enter the first case of the Factorial method. This method checks if its input is equal to 1. If it is, the remaining code in this case is executed, and a value of 1 (1! by definition) is output.

**Figure 4.33: First case of the Factorial method**

If the input value is not equal to 1, the second case is executed. In case 2, the input number is decremented and sent to Factorial again. If the input number was n , the value of $n-1$ would be sent again to factorial, then $n-2$, etc., on and on until the decrementing reached a value of 1. At that point, all of the multiplications would be carried out, yielding $n! = n * (n-1) * (n-2) * \dots * 1$. The factorial is carried out by making the program calculate $n! = n * (n-1)!$. The value of $(n-1)!$ is then calculated as $(n-1) * (n-2)!$, etc.

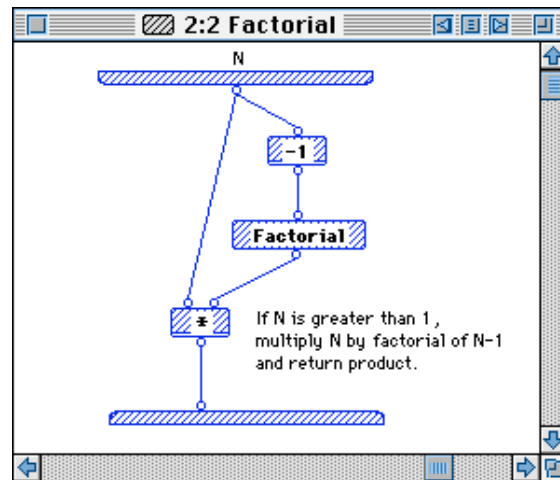


Figure 4.34: Second case of the Factorial method

When the program is executed, the user is asked to enter a number to factorial. The default is 10. Hit return to use this value, or enter a new value.

Figure 4.35: Prompt to number for which factorial is calculated

The factorial of the number is then calculated and presented to the user.

Exercise 4.5:

Write a program to perform exponentiation of a number using recursion instead of the power primitive. Create a method called Recursive Power. Keep in mind that

$$4^7 = 4 * 4^6 \quad \text{and} \quad 4^6 = 4 * 4^5 \quad \text{etc.}$$

The program should ask for both the *base* (the number to be exponentiated, or the number 4 in 4^7), which should default to a value of 2, and the *exponent* (the power, or the 7 in 4^7), which should default to a value of 5. This means that if the user hits the Return key on both prompts, the answer would be 2^5 or 32.

Summary

In this chapter we examined several constructs of Prograph that are responsible for redirecting the flow of control of the program code:

- The ***match*** construct is equivalent to an *if-then*, *if-then-else* or *switch* statement of other programming languages. It executes one or another block of code depending upon the value of a conditional statement. These blocks of code are displayed in separate *case windows* for easy interpretation by the programmer.
- ***Controls*** alter the logical test performed by a match test so that the next case is executed when the match either *succeeds* or *fails*. The rest of the method containing the match is either executed (*finish* control) or not (*terminate* control).
- The ***inject*** node allows the name of a method (and therefore which method to execute) to be determined entirely at run-time.
- ***Recursion*** is a programming technique in which a method “calls itself” repeatedly until a given condition is met. It is useful for certain types of repetitive operations such as factorials, exponentials or power series, which are commonly used in mathematics and engineering.