

3. Prograph Methods

Overview

In this chapter, we'll progress from simple programs to larger, more complex programs using the program elements described in the previous chapter. This chapter and those that follow it in this section of the book will focus upon *structured procedural programming* -- breaking down a large programming task into a *series of more and more focused methods to be executed in turn*. This approach is required as background to the discussion of object-oriented programming (OOP) in subsequent chapters, since even a well-organized object-oriented program must contain some procedural code within it. As we progress through these two sections of the book, we'll take advantage of Prograph's input-output primitives (such as `ask` and `show`) to get information in and out of a program rather than creating a standard user interface. After we've discussed OOP, we'll examine the use of OOP in building sophisticated user interfaces for our programs.

Methods and Operations

To get you comfortable creating programs with the Prograph CPX environment, let's build a program containing a single universal method that tackles a simple trigonometric problem. Given the length of one leg of a right triangle and the angle of one corner of the triangle, we'll calculate the length of the other leg using the *tangent* of the angle. The tangent of an angle is defined as the length of the *opposite* leg of the triangle divided by the length of the *adjacent* leg of the triangle ($\tan \theta = x/y$, where θ is the angle, x is the length of the opposite leg and y is the adjacent leg). Therefore, by rearranging this equation, we can calculate the length of the adjacent leg of the triangle by the equation $y = x \cdot \tan \theta$.

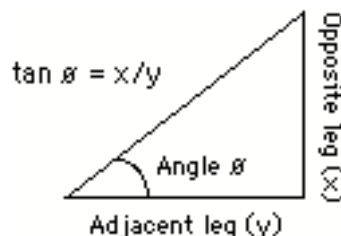


Figure 3.1: Calculating the length of the adjacent side of a triangle with the tangent of the base angle

Begin by creating a new program called Triangle Geometry Project with one section called Triangle Geometry. Next, create a universal method called Adjacent Side in that section, and comment it appropriately (see Figure 3.2). The Prograph editor lets you comment almost any icon or operator in a program, which encourages you to document your programs well with a minimum of effort. With these comments and Prograph's easy to read dataflow diagrams, it should be simple enough for you to understand what your code's doing, even if you're examining it many months later. This

is much different from other languages, such as C++, where commenting is a tedious and active process -- unless you comment profusely, you may never remember what the code means.

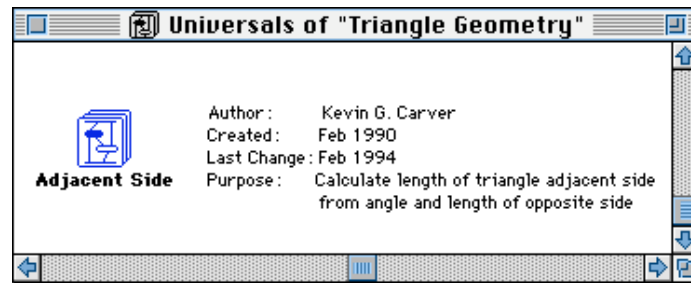
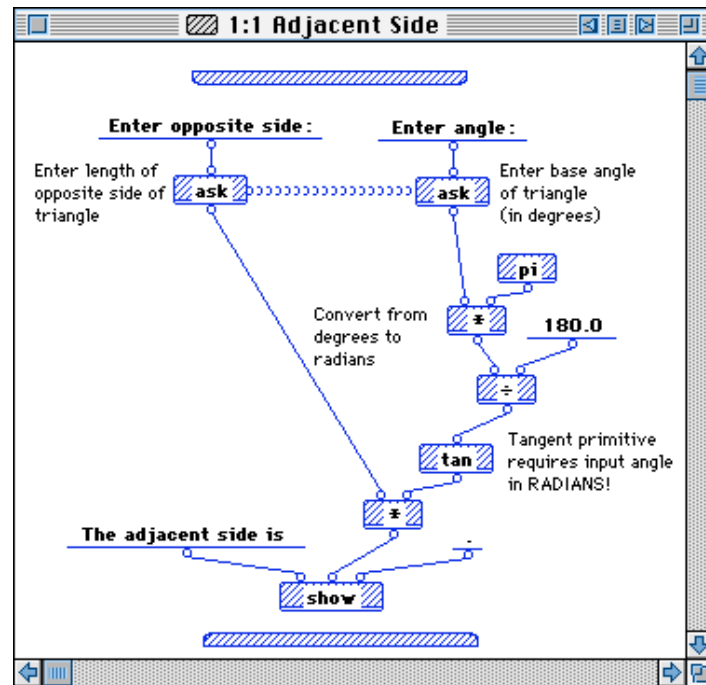


Figure 3.2: The Adjacent Side universal method

Open the Adjacent Side method's window (that is, its first "*case window*") and complete its code diagram as shown in Figure 3.3. Also shown in Figure 3.3 is the equivalent source code in the C++ programming language. Here you can see just how easy it is to understand Prograph code, and just how carefully you have to examine textual code than carries out the same actions. Figure 3.3 also demonstrates how much more natural dataflow programming is than the control flow style of textual languages. Reading the C++ code, we first have to declare the names and storage types of all the variables. Then we must figure out what all the stream input-output code is doing. Next the triangle base angle is multiplied times pi and divided by 180, then reassigned to itself. The tangent of this value is calculated, then multiplied by the opposite side's length. Throughout this code, we must remember what each variable is holding -- their names and types are critical to both our understanding of the code and its correct compilation and execution.

In Prograph, it's easy to see that the base angle is multiplied by pi, then divided by 180. The tangent of this result is calculated and multiplied by the length of the opposite side. Throughout this code, we never have to declare or name any variables -- Prograph keeps track of all of this for us.



```
#include <streams.hp>

const double PI = 3.1415927;

void
AdjacentSide( void )
{
    double oppSide, adjSide, angle;

    cout << "Enter opposite side:";
    cin >> oppSide;
    cout << "Enter angle:";
    cin >> angle;

    angle = angle * PI / 180.0;
    adjSide = tan( angle ) * oppSide;

    cout << "The adjacent side is " << adjSide <<
    "\n";
}
```

Figure 3.3: The Adjacent Side method case window and its equivalent in the C++ language

Now we'll combine the operations on the right side of the window into a *local method* named Degrees To Radians. This local method will now handle the conversion of the angle from units of degrees to units of radians. A radian is 0.01745329252 degrees

(the value π [pi] divided by 180), since a circle is divided into either 180 degrees or π radians.

In Figure 3.4, it's now even easier to understand the purpose of the code with the local method in place. The set of operations now in the local had one purpose -- to convert the angle's units of measurement from degrees to radians. By calling the local "Degrees To Radians," it's made obvious what that code is for. Using locals provides one more way to document the purpose of our code.

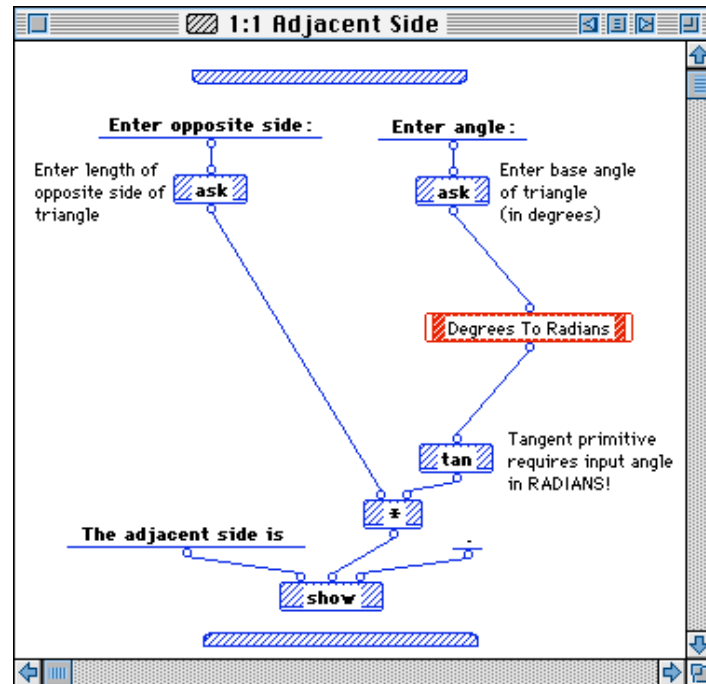


Figure 3.4: The Adjacent Side method case window with a local method

The contents of the local method are still there. You can view them by opening the Degrees To Radians case window (see Figure 3.5). What's especially convenient about local methods is that they incur no execution overhead; that is, the code runs just as fast as it would were the local not defined.

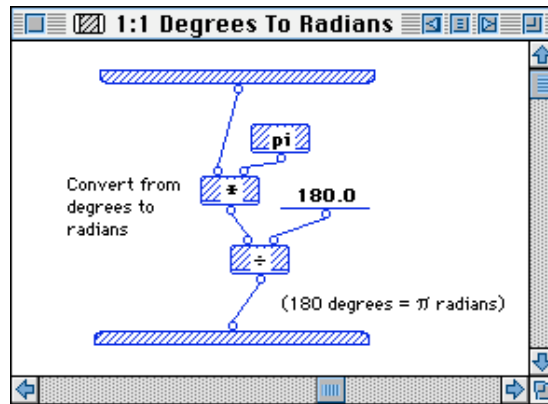


Figure 3.5: The Adjacent Side method case window

Remember that in dataflow programming, the order of execution of operators is determined not by their position in the code window, but by when all of the input data is made available to a given operator for it to work on. When the Adjacent Side method starts, Prograph has the choice of executing either of the two *ask* primitives at the top of the code diagram. Which one is executed first doesn't matter to the Prograph interpreter. But in this case, the order of execution *does* matter to *us*. We want to execute the *ask* operators in a particular order. We can force the interpreter to execute the leftmost *ask* first by means of a

synchro link, as shown in Figure 3.6. Synchros are a means for you to force code to execute in a *control-flow* manner as in other languages like C -- the operations on each side of the synchro must execute in the order you define; that is, the order of execution is no longer data-driven. The operations linked by a synchro execute in the order depicted by the direction in which the semicircles that make up the synchro "point." In Figure 3.6, for example, the semicircles of the synchro "point" from the left to the right, so the *ask* primitive on the left will execute before the *ask* on the right.

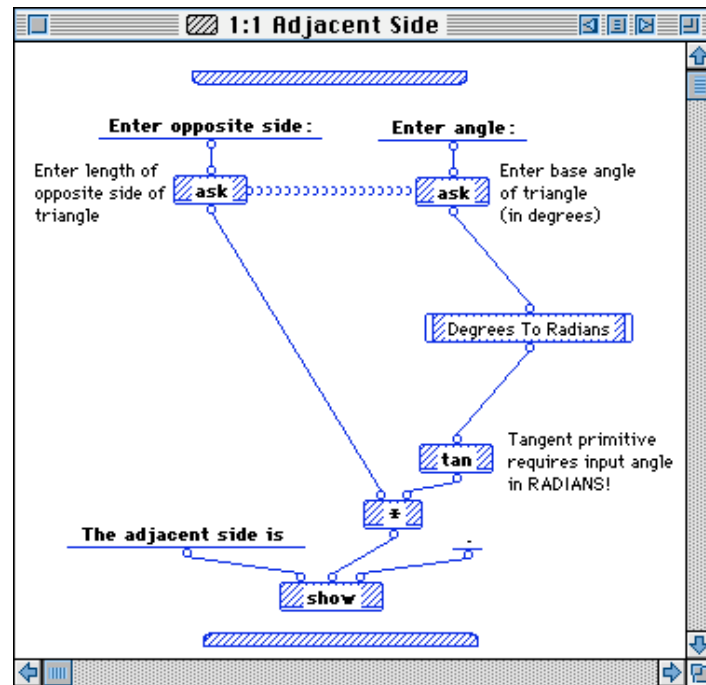
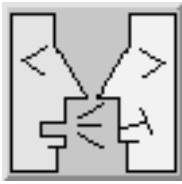


Figure 3.6: Forcing the order of execution with a synchro link

The program now must first ask for the length of the opposite leg of the right triangle. This value is fed to the first root node of the division \div primitive but this primitive cannot execute until it also receives the input data on its second root node. This input is the product of the code on the right side of the window, which starts with the right **ask** primitive at the top of the code diagram. The next step of execution is therefore that we are asked to enter the angle at the corner of the triangle.

Following this, we will calculate the tangent of this angle and multiply it by the length of the opposite leg to get our answer -- the length of the adjacent leg of the triangle. Computers, like hand-held calculators, have built-in trigonometric routines. Unfortunately, while most people think of angles in terms of degrees, these computer routines expect angles in units called *radians*. The tangent primitive (**tan**) is no exception. We must convert the angle from degrees to radians if the **tan** primitive is to give us the output we desire. The local method Degrees To Radians was created to contain the code to do this. All this local method does is multiply the angle in degrees by π or pi (obtained from the **pi** primitive) and divide it by 180, yielding the angle in radians (radians = $\frac{\text{degrees} \times \pi}{180}$). (Note: converting from radians to degrees is accomplished by the opposite steps -- multiplying by π and dividing by 180.)

After the conversion from degrees to radians is finished, the rest of the code will function properly, and our answer will be correct. To see for yourself, execute the Adjacent Side method by highlighting its icon, then selecting the Execute Method menu item in the Exec menu.



By The Way...

You will use methods to convert from degrees to radians, and vice-versa, fairly often. You may wish to create universal methods to do these conversions and save them as a single section called “*Angle Conversions*”. You can then *reuse* these methods over and over again. Remember, the more you can reuse code, the less time you’ll spend programming later. You can convert a local method into a true universal method by highlighting the local method’s icon, then selecting the *Local To Method* item in the *Ops* Menu.

Exercise 3.1

Create a program called Calculate Angle to ask for the length of two sides of a triangle and the angle enclosed between them. Compute the length of the third side of the triangle with the Law of Cosines:

$$a = \sqrt{b^2 + c^2 - 2bc(\cos A)}$$

where b and c are the sides of the triangle and A is the enclosed angle. Use the `sqrt` primitive to find the square root.

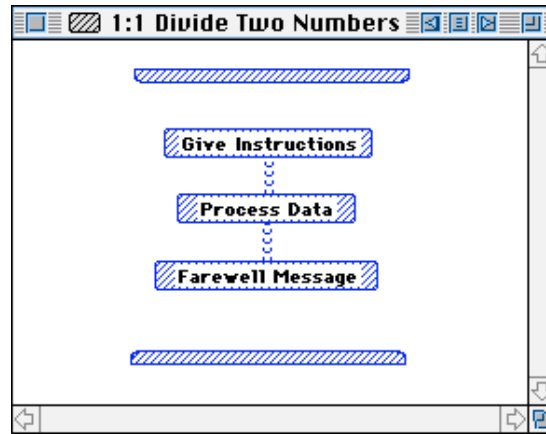
Structured Programming of Bigger Programs with Many Methods

If a program is to do any meaningful tasks, it must include a number of methods. This is where the beauty of Prograph comes through! Prograph encourages *structured* procedural programming -- the planning and construction of a program in a *top-down* manner, breaking apart a huge complicated task into smaller, well-defined, manageable tasks, then breaking down those tasks even further, and so on. In structured programming, writing a program starts by first determining the *major* tasks you want the program to accomplish (the “top” or more *general* level of the program), then fleshing out the details of each of the minor steps that allow the big steps to get done (filling in the “bottom” or more *specific* levels of the program). Each task is broken down into as small and as specific a sub-task as it can be, and so on. The Prograph interpreter can help you plan your programs in this fashion by letting you start programming the top-level general aspects of the program, then automatically creating the bottom-level methods for you.

Let’s look at a concrete example. Our next program will display instructions to the user, then ask for two numbers, divide them, print the results, and sign off with a farewell message. Create a new project called Divide Two Numbers Project, with a section and universal method each having the name Divide Two Numbers, then open the universal method’s case window.

We’ll start building the program by defining the general broad steps the program should take. At this point we will disregard the details of how to accomplish these steps. Create three operations, one above the other, named Give Instructions, Process Data and

Farewell Message. Connect the operation icons with synchro links to ensure their order of execution. Your Divide Two Numbers case window should look like Figure 3.7.



```
void GiveInstructions( void );
void ProcessData( void );
void FarewellMessage( void );

void
DivideTwoNumbers( void )
{
    GiveInstructions();
    ProcessData();
    FarewellMessage();
}
```

Figure 3.7: The top-level Divide Two Numbers method and its C++ language equivalent

Notice in Figure 3.7 that in Prograph you don't have to declare your methods before calling them as in the C++ code also shown. Declaring methods is a necessity in languages in which you must explicitly state the type of inputs and outputs a method will have. In Prograph, all of this is handled automatically for you -- just one more example of how Prograph simplifies programming by minimizing tedious and error-prone "housekeeping" chores. In a large program, this can save a lot of time and effort.

It's important to note that we haven't actually *created* the methods called Give Instructions, Process Data or Farewell Message yet. We've just told Prograph that *we will be calling methods with these names* in our program. That is, we've planned out the general steps of our program, but haven't yet written the specific code to accomplish these steps.

Now that we have the top level of our program and its general flow of instructions, we'll let the Prograph interpreter help us fill in its details. Close this case window, then select the Divide Two Numbers method icon in the sections window, and

execute the method. What will happen now? How will the Divide Two Numbers method execute when the three methods it should call do not exist yet?

After it starts executing, the interpreter tries to call the first method to be called by Divide Two Numbers -- the Give Instructions method -- but can't find it since it doesn't exist. It displays an alert message to tell you so (see Figure 3.8). But there's one important part of the alert that's easy to miss. The alert gives us the option of *creating this missing method*.

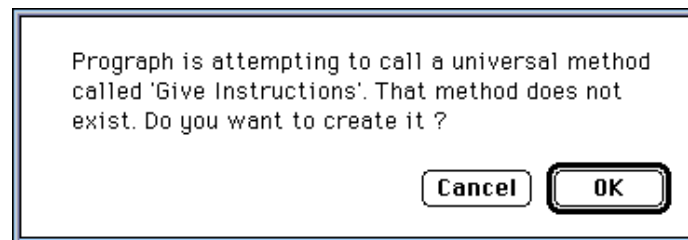


Figure 3.8: Method Does Not Exist alert

If we click on the OK button of the alert, the Prograph interpreter will create the Give Instructions method. A dialog box, shown in Figure 3.9, will pop up to ask us in which section to place this new method. At this point, our program only has one section (also called Divide Two Numbers), so let's select that one.

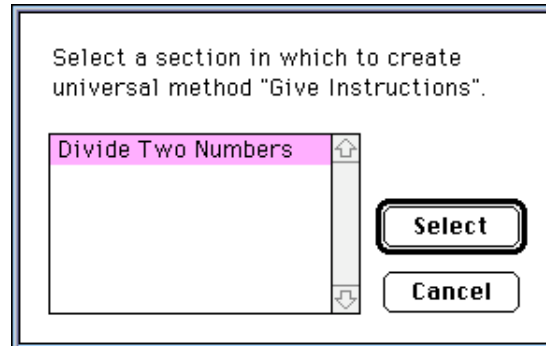


Figure 3.9: Dialog for selection of section in which to place new method

A new case window (Figure 3.10) opens with the name of the newly created method -- Give Instructions -- in its title bar. The stippled background of the window tells us that this method is now being executed. We've actually created a new method *while the program is executing*! The created method is given the proper number of inputs and outputs (in this case, none).

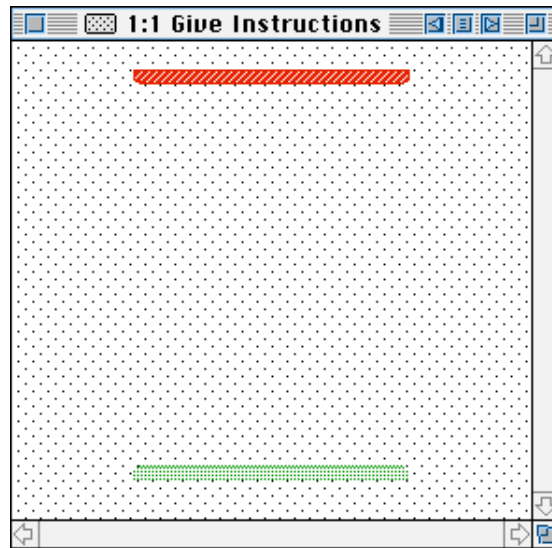


Figure 3.10: Creation of new method during runtime

To continue execution of the program, select Resume “Divide Two Numbers” in the Exec menu. As the program continues to execute, alerts, dialogs and case windows similar to those of Figures 3.8 to 3.10 will be presented for each of the two other as yet undefined universal methods, Process Data and Farewell Message. Continue to execute the program until these methods have also been created. If you now open the Universal Methods window for this section, you’ll find in addition to the Divide Two Numbers method the three new methods that we’ve created by running the skeleton of our program.

We’ve started the construction of our program automatically. All we did is create the top-most level of our program -- the Divide Two Numbers method. Then we defined the most general steps the program would take -- presenting instructions, processing some data, then presenting a farewell message -- and gave descriptive names to these steps. By attempting to run the program, the interpreter filled in the missing methods, providing the next lower level of the program. We can now write code for the newly-created methods or just leave them empty for now until we’re ready to define their code. This ability to call methods that don’t yet exist and create them “on the fly” allows us to *plan and test the program as we write it*, a process called *prototyping*. Writing and testing a program as you go gives you immediate feedback about whether the program design really matches what you intended to do, before you progress to the point where changes become painful to accomplish. Prototyping with Prograph encourages you to experiment with your code and try out different ways of carrying out each program step. So long as the different versions of the method you’re experimenting with maintain the same number of inputs and outputs, they can be interchanged without “breaking” the rest of the program.

Let’s continue to refine our program by proceeding to its more specific code. In other words, we’ll start to define exactly how each of the major steps of the program will be accomplished by breaking them down into the individual actions needed to do perform

each major task. Start with the simple Give Instructions method. Create the code diagram shown in Figure 3.11 which will perform the actual procedures to display instructions to the user. In this case, the steps required are so brief that they can be accomplished entirely within this method alone. The Change Value dialog is also shown in the figure for the text constant fed into the show primitive so that the entire instruction set can be viewed in its untruncated form.

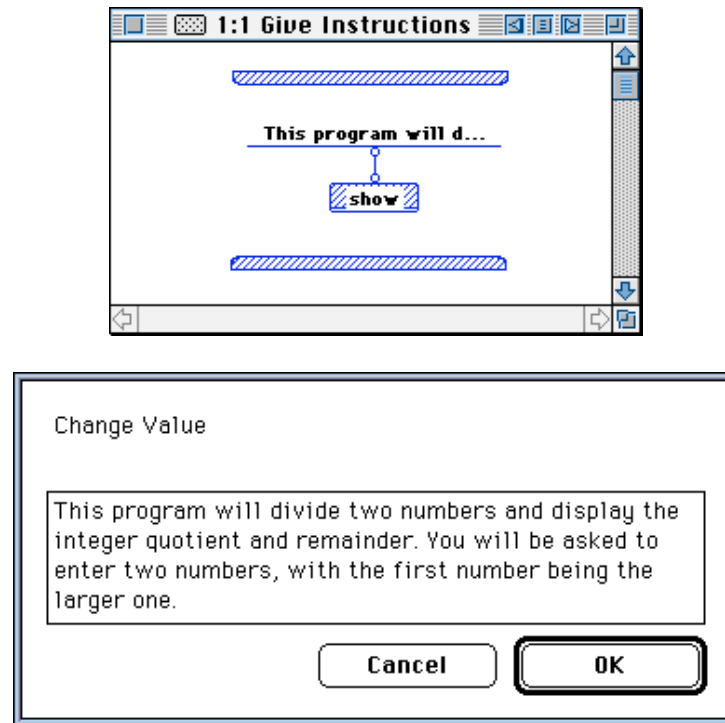


Figure 3.11: The Give Instructions method and its text message

Repeat this process with the Farewell Message method. Its case window should wind up looking like Figure 3.12.

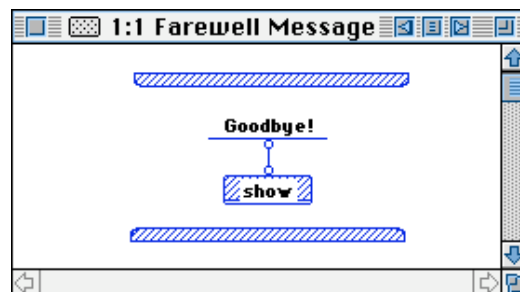


Figure 3.12: The Farewell Message method

Let's progress to the middle method, Process Data. Our goal for this step of the program is to get two numbers, divide them, and display the answer. We can therefore

continue to “subdivide” this step into the three actions we wish to accomplish -- enter two integer numbers, divide them, and display the results of the division. These substeps define the next, more specific level of our top-down program design. We’ll name three methods to be placed within the Process Data method -- Get Integers, Divide Integers and Display Answer. Fill in the case window as shown in Figure 3.13. This provides Process Data with the names of the three methods it will call.

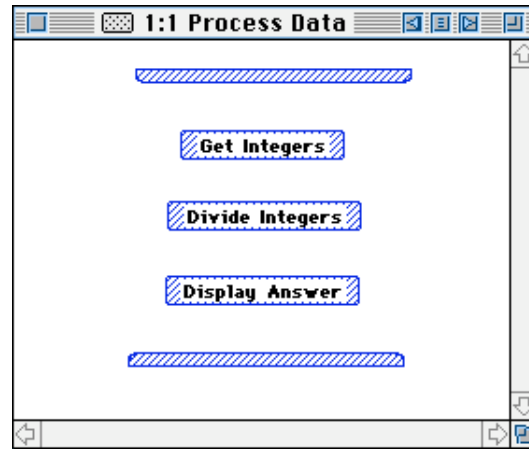


Figure 3.13: The steps taken by the Process Data method

We now write the code to perform the actions at this next level of the program. This time, rather than have the interpreter create the three methods to be called by Process Data for us, let’s create them ourselves. Open the Get Integers method’s case window and create the code diagram shown in Figure 3.14 for this new method.

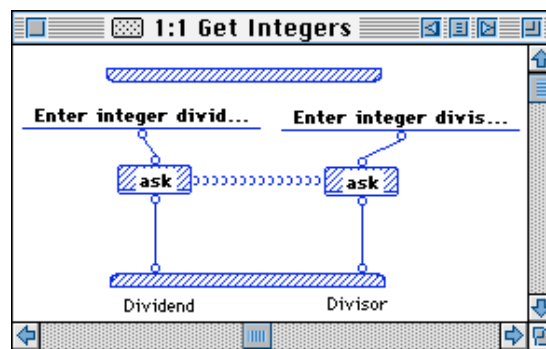


Figure 3.14: The Get Integers method

Notice that we feed the two numbers entered into output nodes on the output bar of the method. These numbers will become the outputs of the Get Integers method -- the number to be divided (*dividend*) and the number by which to divide it (*divisor*). Close the Get Integers window. The Get Integers method root nodes, along with their associated comments, are shown in Figure 3.15.

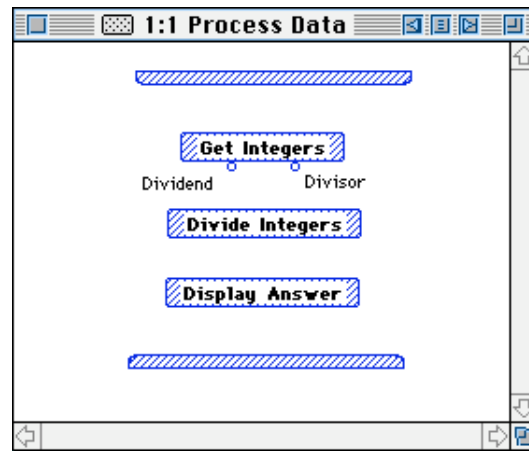


Figure 3.15: Nodes and comments of the Get Integers method

Continue by creating the code diagram for the Divide Integers method (Figure 3.16). The $\div\div$ primitive performs an *integer* division, producing two outputs -- the quotient of the two numbers and the remainder.

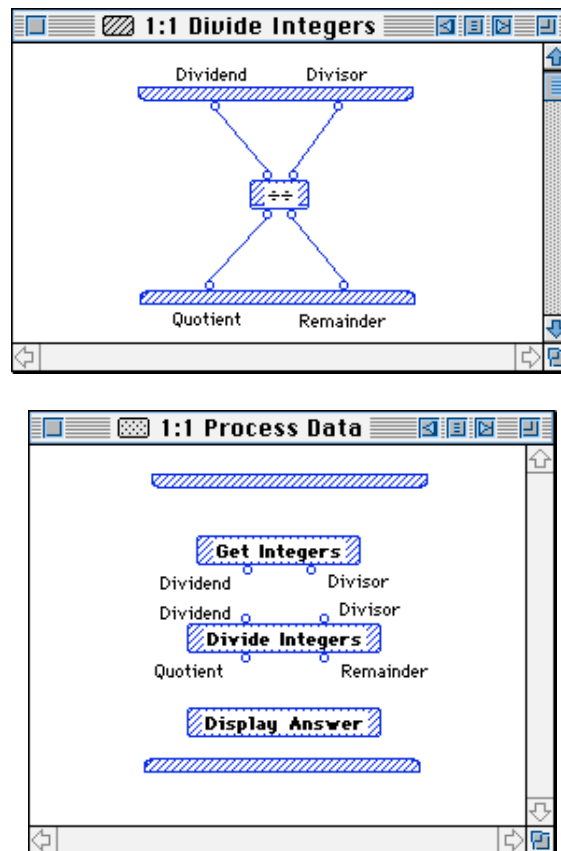
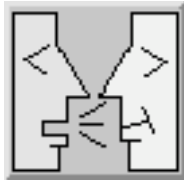


Figure 3.16: The Divide Integers method



By The Way...

Every Prograph primitive and user-defined method has a specific “arity”. The arity is the number of input and output nodes. If you try to access the method with the wrong number of inputs or outputs, you’ll get an arity error message when you attempt to run your program. Always double-check that you’re calling methods using the correct number of inputs defined for that method. How can you do this? Prograph provides on-line help even for methods *we* write. Every time you write a method, it’s added not only to Prograph’s table of available methods, but also to the Info... window’s list of methods. We’ll show you how this works in the next section. Once the information is in the help system, you can use the Info... window to look up your method’s purpose, inputs and outputs.

All that remains to do is create the Display Answer method, whose code diagram is shown in Figure 3.17. You might notice that the `show` primitive icon in the figure is wider than usual. The icon is widened by dragging its terminal nodes laterally. The farther apart you move the nodes, the wider the icon becomes. This makes the icon wide enough to neatly display all of the extra nodes we’ve added to it.

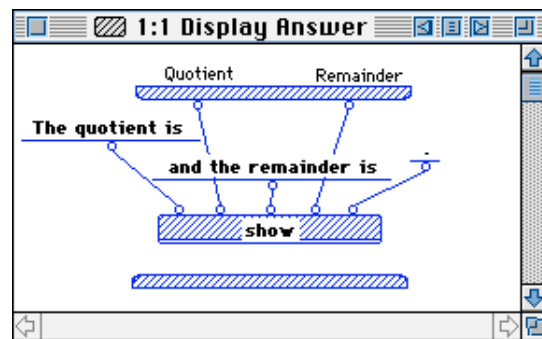


Figure 3.17: The Display Answer method

Finally, return to the Process Data case window and connect the root and terminal nodes of the method icons so that the data required by each method will flow between them (see completed Process Data method in Figure 3.18).

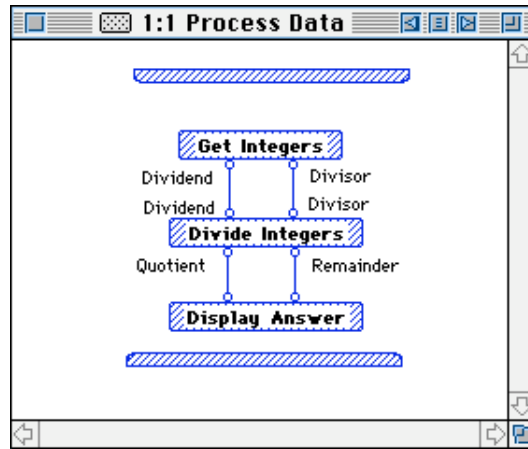


Figure 3.18: The completed Process Data method

We've now designed a more complex Prograph program using structured procedural programming's philosophy of top-down design. We started at the highest level of the program -- the general tasks that the program had to do. We then defined those general tasks in three methods -- Give Instructions, Process Data and Farewell Message. Next, we progressed to the next lower levels -- how these methods' tasks are accomplished. For the Process Data method, we broke down the problem of processing the data into three smaller tasks, defined in three additional methods -- Get Integers, Divide Integers and Display Answer. The overall design for writing the program is pictured in Figure 3.19. The program is decomposed into three general methods, and one of these -- Process Data -- is further broken down into three more specific methods.

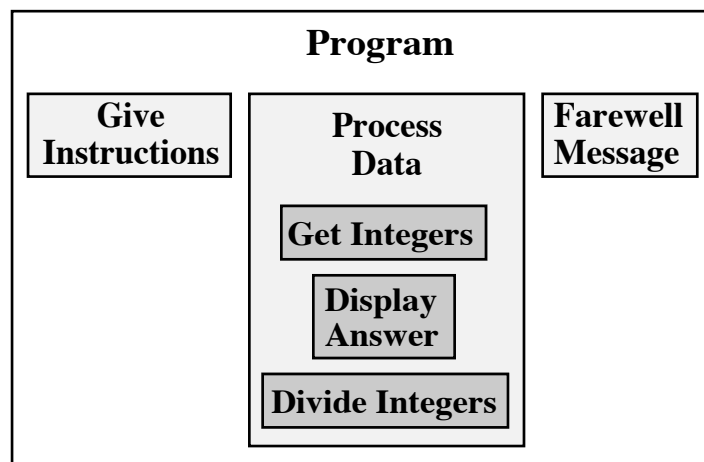


Figure 3.19: Organization of the Divide Two Numbers program

The top-down design helps us define the nature of the problem we want to solve and break a large problem down into smaller, more manageable tasks. It also lets us build a program with code that will be easier for us to understand at a later date.

Exercise 3.2:

Create a Gasoline Costs project. Write a small program that will start by providing instructions to the user, and end by presenting a *Thank You* message to the user. The main task will be to calculate the gasoline consumption of a car (in miles per gallon) and the fuel cost for each mile traveled. The user should be asked to enter the starting mileage, ending mileage, number of gallons of gasoline purchased and cost of the gasoline per gallon. Use proper procedural programming style.

Setting Detailed Help Information for Your Methods

We've written several methods in Divide Two Numbers. While we can easily add comments to each method to describe its purpose, wouldn't it be great if we could get the Info... on-line help system to display information about our methods as well?

Bring up the Info... window and display the information about the Get Integers universal method that we just wrote. All that appears in the help window is the name of the method. This is not very useful. What we'd like to see as well is what input data the method expects, what output data it produces, the data type of each input and output and what the method does. We need to add this information to the help display.

Open a new text-editing window by selecting the New Text item in the File Menu. Don't worry about giving the text window a name when you're prompted for one, since we'll discard this window when we're finished with it. Type the information shown in Figure 3.20 into the window.

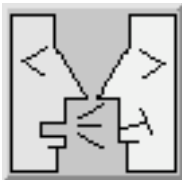
Inputs: none

Outputs: integer; integer

Prompts user for two integer values.

Figure 3.20: Comments to be added to the Get Integers method

Highlight all of this text, then copy it. You may now discard the text window. Now find the Get Integers icon in the Universal window and proceed to add a comment to it. When the comment prompt appears, paste the text you copied from the text window into the new comment.



By The Way...

You could have typed all this directly into the comment of the method, but when you are typing in a lengthy comment, it can be easier to use text windows as we just did instead.

Comments for methods can be displayed in the Info... window. Bring up the Info... window again. When you select the Universal Methods item, you'll see a list of

the universal methods that you created for the current program. When you request information about the Get Integers universal method, the dialog will now display the name of the method, as well as the comments you typed -- the method's inputs and outputs, and what it does.

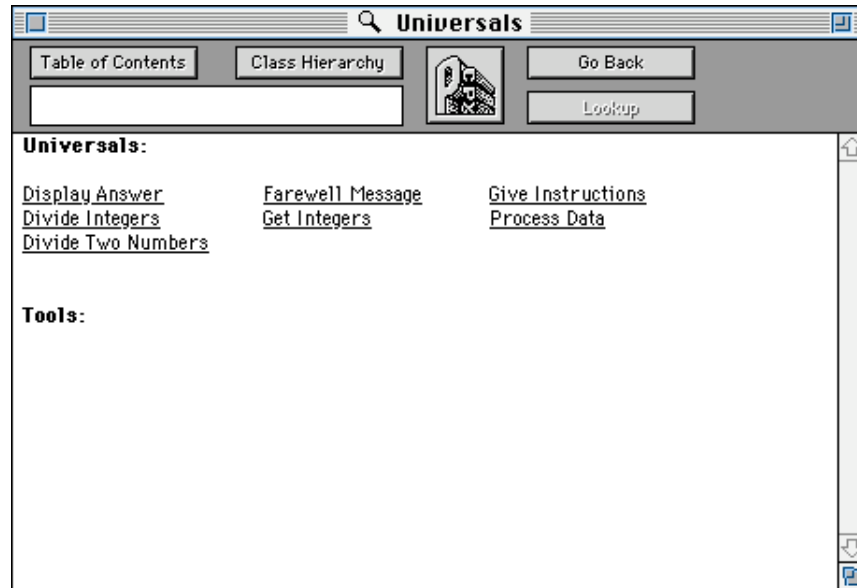


Figure 3.21: Universal methods in the Divide Two Numbers program

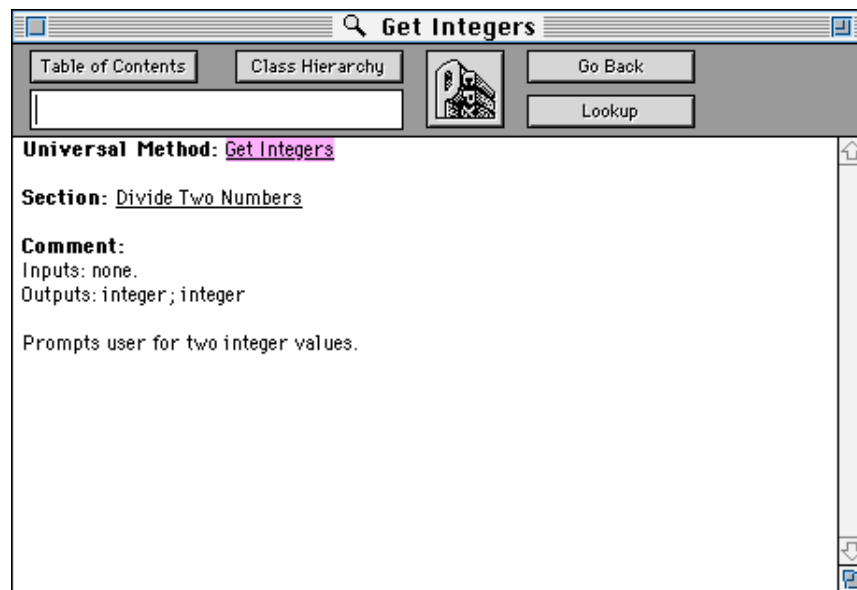


Figure 3.22: Information about the Get Integers method

We suggest that you get into the habit of adding similar comments to all of the methods you write so that they present similar information as Prograph primitives, external methods and classes in the Prograph help facility.

Summary

- **Methods** are the building blocks of a Prograph program -- independent modules of dataflow code that can be reused and called throughout the program. Unlike other programming languages, Prograph allows you to execute independent methods instead of an entire program, making debugging portions of a program much easier.
- **Local methods** are a way to bundle functionally-related portions of code into a “bundle” that can be accessed only within one method. Their chief purpose is for making code more understandable, but they are also useful for logical tests, as we’ll see in code examples throughout this book. Local methods incur no run-time overhead, so feel free to use them often.
- The ability of the Prograph interpreter to run incomplete code encourages **prototyping** of structured code and experimentation.
- Just by writing comments for a method that you’ve written, information about the method is automatically added to Prograph’s **on-line help facility**.