

2. Elements of a Prograph Program

Overview

We have already demonstrated some of the elements that make up Prograph dataflow code diagrams and programs. Before we examine Prograph programming in detail in the upcoming chapters, it would be helpful to discuss the basic building blocks of a Prograph program, and relate them to programming constructs in a more conventional programming language, in this case, the C or C++ programming language. *It is not necessary for the user to be fluent in the C or C++ language to learn the Prograph programming language; we simply provide examples of similar constructs in these popular languages so that experienced programmers who are switching from conventional textual languages will have a point of reference.* We'll discuss the data elements and code constructs of Prograph, as well as the data storage formats for Prograph data. By the end of this chapter, the reader should have some ideas about how and when to apply these elements in building programs.

There are two major types of program elements or components that are pieced together to construct a Prograph program -- *data* and *operations*. Let's examine operations first.

Operations

Operations are the basic steps taken by a Prograph program. An operation that has been created by you in a code window, but has not yet been defined as a specific type of operation is called a "*plain*" operation. This *generic* operation serves as a "place holder" in your program until you define *what type* of operation it should be.



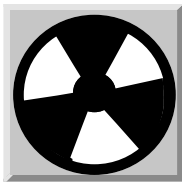
Figure 2.1: An as yet untyped or "plain" Prograph operation

Operations and Methods

Methods are operations that form the fundamental building blocks of a Prograph program. They are modules or packets of instruction code that can be reused in several places in a program. Methods are much like C++ functions, but represented in a graphical chart-like manner rather than as text. However, unlike flowcharts, in Prograph the flow is not of one instruction to the next, but rather a flow of *data* from one program step to another. In other words, the program execution order is determined not by what instruction was written next, but whether or not data has arrived at that step.

Operations are used to "*call*" a Prograph method; that is, to tell a method to execute itself. When you first create an operation (a method call), it appears as the

“plain” Prograph operation shown above in Figure 2.1 until you give it a name. After you type in its name, the Prograph editor checks that name against its database of existing methods in your program, whether they’ve been supplied as part of the Prograph environment or designed by you. If it finds an existing method whose name matches the name you’ve given the newly created plain operation, it converts the plain operation icon into an *operation* icon that is characteristic of the *type* of method that it calls, giving you immediate visual feedback of what the operation does. The operation icon is also provided with the proper number and type of inputs and outputs for the method it will call. In textual programming languages like C++, all you’d see in the source code is a name or label, which could reflect any type of code or data -- function, variable or constant. It would be up to you as the C++ programmer to remember what that label meant -- whether it represents code or data -- as well as what inputs and outputs, if any, such code expected. If you didn’t remember correctly, you’d get a program compilation or execution error. Prograph, with its visual feedback and method lookup system, helps to keep you from making these potential mistakes.



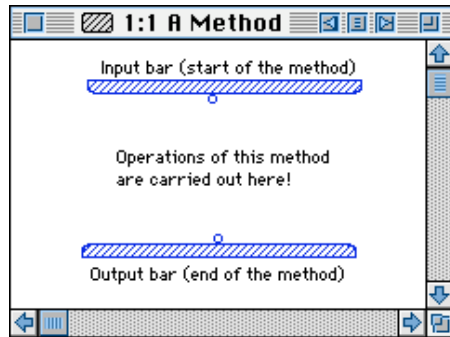
Warning!

As in the C++ programming language, names of operations and data elements are *case-sensitive*! That is, the name “My Method” is not interpreted by Prograph to be the same as “my method” or “My method.” Be *very careful* with punctuation to *exactly match* the name of an operation, method or data element you are accessing again.

Prograph programs are typically composed of several operations or method calls nested within other methods linked together to form a larger, more complex dataflow chart. While this is also true of other languages like C++, there is one less obvious advantage of Prograph’s linked method modules. In C++, you are forced to build a large part, if not all, of a program before you can execute the program to see if any one method works correctly. In Prograph, methods are independent modules that may actually be executed *by themselves*. This makes it much easier to debug a Prograph program, since you can run and debug it piece by piece, noting if each method works correctly in isolation before checking its integration into the entire program.

Methods contain three basic parts: an *input bar*, an *output bar*, and the *operations* the method will carry out (see Figure 2.2). When a method is first created, its code window is empty except for the two bars at the top and bottom of the window. The input bar at the top of the method’s window denotes the *start* of the method, and the output bar at its bottom denotes its *end*. At this point, we can add operations to our method, but we can’t send data to the function or get data back from it. This is what the two bars are really for. If we create a node (that small dot in the figure) on the top bar, we’ve told the method to expect one piece of *input data*. Likewise, we can create nodes on the bottom output bar, we’ve signaled the method to provide one piece of *output data* when it finishes executing that will most likely be sent to another operation. This clearly labels the inputs and outputs of the method.

Data flows into a method through the nodes on its input bar, then flows one by one through operations within the code diagram that either examine or modify that data, then finally flows out of the method by nodes on the output bar. Programming entails defining and “wiring” the paths of the data flow -- nodes, datalinks and operations.



```
OutputType
AMethod( InputType inputValue )
{
    OutputType outputValue;

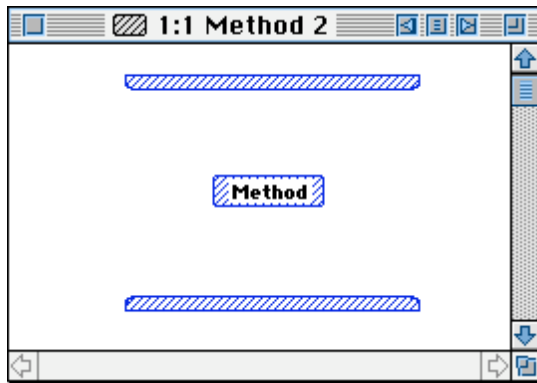
    // Operations carried
    // out here

    return( outputValue );
}
```

Figure 2.2: The components of a Prograph method compared to those of a C++ function

The visual representation of the method’s inputs and outputs serves another function. Once we define the number and types of inputs and outputs for the method, this information is added to the Prograph environment’s table of existing methods. If we try to call this method in another place in our program, the correct number of inputs and outputs will be attached to the called method’s operation icon for you.

There are several types of methods in Prograph. Each type has a different operation icon type to help you easily differentiate them. The most basic type of method is the *universal method*. This is a general method that can be used *anywhere* in a program. Universal methods are not associated exclusively with object-oriented programming (these so-called “*class methods*” will be discussed in Chapters 8-11). Calls to universal methods appear in our code diagrams as a plain box symbol containing the name of the method to be executed. The universal method named Method 2, shown in Figure 2.3, expects no inputs or outputs, so it has no root or terminal nodes. It contains one operation -- a call to another method named Method, which expects no inputs and provides no outputs.



```
void
Method2( void )
{
    Method();
}
```

Figure 2.3: A Universal method with no inputs or outputs (itself calling a second method with no inputs or outputs), shown with its C++ equivalent

A second type of method is the *primitive method* or *primitive* for short. Primitives are just like universal methods, but they have been written for us by the folks at Prograph International. A library of pre-written methods called *primitives* are packaged with Prograph CPX. Some primitives serve a similar purpose as the proposed ANSI standard C++ library, but the primitives also include machine-specific methods and methods that simplify complex tasks that would otherwise require calls to the computer's operating system.

To differentiate a call to a primitive from a call to a universal method, a primitive's operation icon is a box with a *single horizontal line* added along its bottom edge. The shape of the icon gives you immediate recognition of whether the operation represents code, data or a combination of the two (class). All *methods* or code (whether written by the user or provided with Prograph) have a *rectangular box* for their operation icon. The details of a method's box-shaped operation icon, such as the presence of a line running parallel to its bottom, helps differentiate the *subtype* of method the operation calls (universal method, primitive, etc.).

Primitives have names that tend to begin with lower case letters. In Figure 2.4, we see a primitive called `show`, which displays text in a dialog box. It's similar in function to the C++ `printf` or stream output functions.



```
printf( ... );
or
cout << ... ;
```

Figure 2.4: A primitive method call and its C++ equivalent

You can also see in Figure 2.4 that there is a dot or *node* on the top of the `show` primitive's icon. The node represents a route for data to flow *into* the method called by the operation, in this case the text to be displayed by the `show` primitive.

Prograph allows you to access routines written in other programming languages as *external methods*. These look like icons that call universal methods in a code window, except that they have *two horizontal lines* -- one at the top and one at the bottom of the operator icon. For example, the icon in Figure 2.5 calls the Macintosh operating system routine `SetPt`, a routine written in the Pascal language that sets the screen coordinates of a point to be moved to or drawn on the screen of a Macintosh computer. Similar external routines are provided in the Windows version of Prograph.

External code can also be written to optimize the execution speed of time-critical portions of a program. For example, if you really need to squeeze every last bit of speed from a given Prograph method you've written, you have the option of rewriting it in the C language and linking it into your Prograph program with the optional Prograph C Tools kit from Prograph International.



Figure 2.5: An external method call that accesses the SetPt Macintosh Toolbox routine written in the Pascal language

The last type of method we'll mention for now is the *local method*. Its operator icon is a box with *vertical lines* on the left and right sides. These are used solely for the purpose of making our program code more legible. Let's create one to show what it is. Figure 2.6 shows a method called Multiply Numbers. There's a lot going on in this method, isn't there?

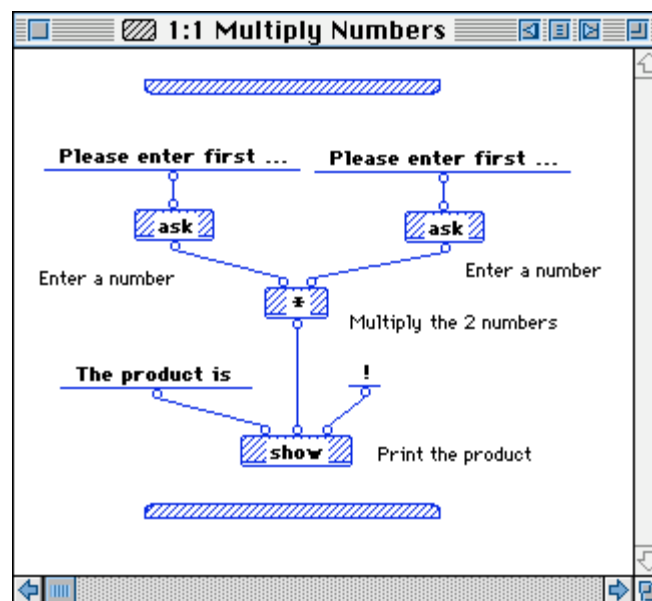


Figure 2.6: A crowded method case window

Let's clean up the Multiply Numbers method window a bit. First, we select the two **ask** primitive icons, along with their text constant inputs. Then we select the **Oper**s To Local menu item. All of the selected operation icons are now replaced with a *single* icon. We'll name this icon Get Two Numbers. The simplified Multiply Numbers method is shown in Figure 2.7. The Multiply Numbers case window is now much simpler to understand at a glance.

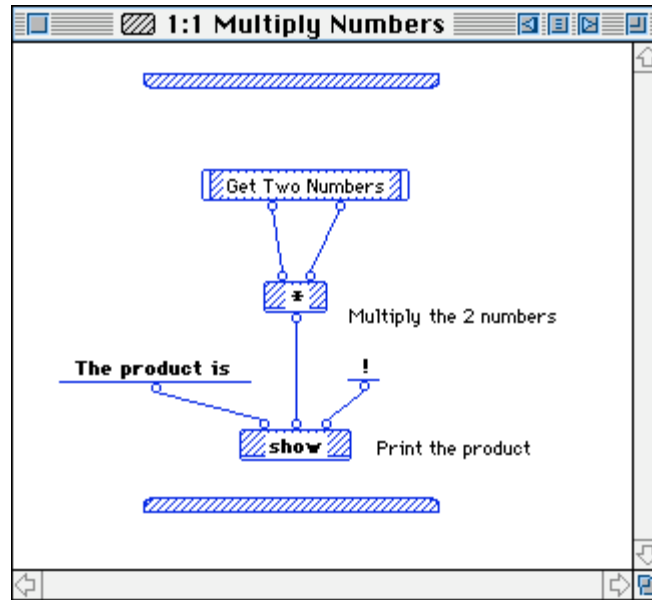


Figure 2.7: Creation of a local method to simplify method code

What happened to the **ask** primitives? These operations are still present, but we've neatly stored them away, out of view, to make our Multiply Numbers case window neater and simpler. If you double-click the Get Two Numbers operator icon on one of its click spots, another case window (see Figure 2.8) will open that contains the "missing" commands. All we've done is *group together* these operations and enclose them within a separate *local* method. The term "local" implies that this code is "seen" only by the Multiply Numbers method that contains it. The Get Two Numbers local method *can't* be called by *any other method* in our program.

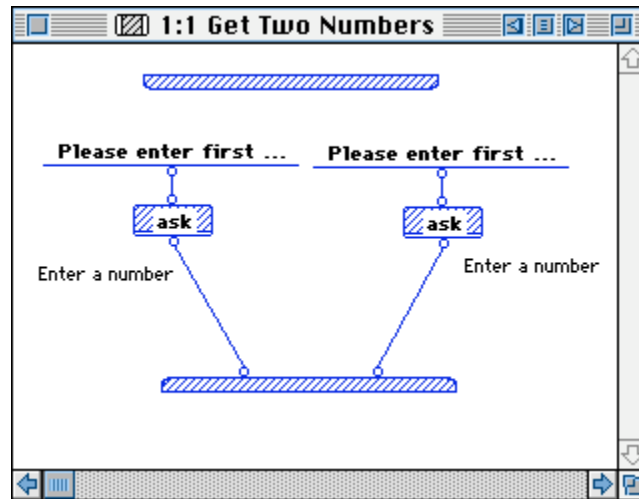


Figure 2.8: Contents of the local method

Notice that the Get Two Numbers local method's icon in Figure 2.7 has no input nodes (terminals), but does have two output nodes (roots). This is because the two **ask** primitives each output a number that the local method will have to output as well (see Figure 2.8). The local method is automatically given the proper number of inputs and outputs. Once again, Prograph takes care of the mundane tasks of programming for us -- we don't have to keep track of details like these.

Information hiding with local methods is a Prograph feature that is not found in languages like C++. In C++, the only option you have for "hiding" code is to make a new function to contain it. When calling this new function, your program will still incur the overhead of a function call (unless it is an extremely short "inline" function, which is executed without function-calling code). An equivalent Prograph local method carries no function call overhead since it isn't really a method, but a convenience to make the program code more understandable by labeling groups of operations with their purpose.

Built-In Methods -- The Prograph Primitive Set

Before discussing other elements of Prograph programs, let's take a quick peek at the collection of primitives that have been packaged with Prograph. The Prograph primitives carry out a wide range of functions that we'll use frequently. In fact, it is extremely difficult to write a Prograph program without using primitives. Some of the primitives execute simple functions, but others are fairly complicated, such as the file read/write primitives, directory primitives, memory primitives and graphics primitives. The primitives therefore save us a lot of programming effort and shield us from having to make some operating system calls. Consult the Prograph Reference manual for a complete description of each primitive.

A subset of the Prograph primitives serve the function of basic C++ language operators. For example, there are mathematical primitives like the ***** or multiplication

operator, and the `+` or addition primitive. In C++, these operations are built into the language syntax. The mathematical primitives build them into the Prograph language. Similarly, logical primitives perform tests on data such as whether the data value is greater than or less than a particular value. They also test logical states -- whether something is true or false -- or whether *more than one* condition at a time is true or false. Other Prograph primitives perform tasks included in the proposed ANSI Standard C++ library -- functions that are not built into the C++ language but tend to be supplied with C++ compilers anyway. An example is the wide range of trigonometric operations such as sine, cosine, and tangent. Other examples are rounding-out and truncating operations, random number generation, and even some financial functions not found in C++.

Perhaps the most important primitives are those used for getting data in and out of the program. We've already used two of them -- `show` and `ask`. These primitives take the place of the C++ language's `cout` and `cin` stream I/O functions, which output and input text or numbers, respectively. Like their C++ equivalents, they are "smart" methods, which can handle a wide range of input data types and print them correctly.

The string primitives are used to manipulate text strings. They allow you to find substrings within a string, join strings together, and convert characters from string data to the values of the ASCII numbers that represent them. In C, most of these functions are served by low-level character manipulation functions. To get the ease of use that the Prograph string primitives offer, you'd have to buy additional string libraries or write them yourself.

Prograph also provides built-in data types that are not a part of other languages, such as *lists*. A rich set of list primitives is provided for convenience in using lists. In C++, lists must be created as a new data type. Several intermediate and advanced books on C++ programming provide code for list processing as an example of a programming task beyond the capabilities of a beginning programmer. Not so in Prograph.

Most of the remaining primitives serve to augment the *Application Builder Class* library that is supplied with Prograph CPX to manage the user interface of your completed application program (we'll discuss them in Chapters 13-15).

One important feature of the Prograph primitives is that many of them are *polymorphic* -- that is, they can accept a wide range of input data types. Languages such as C force you to input a particular data type and only that data type into a function. The C++ language, allowing you to make polymorphic functions via function overloading, saddles the programmer with the chore of setting this up. Prograph, on the other hand, is much more forgiving and avoids more scenarios where errors could occur. Primitives allow you to input different data types without "choking."

Decision-Making with Matches, Controls and Cases

Some of the Prograph primitives give you ways to perform *decision-making* tasks, such as the logical operations, or the input-output primitives such as `answer`, which

gives different outputs depending upon a button selected by the user. As part of the Prograph language, there also exist several operations called *matches* that perform different actions depending upon if a particular condition is true or false. Matches are equivalent to the *if-then* or *case* statements of C++. For example, this match checks if the data value it receives on its root node is *equal to zero*:

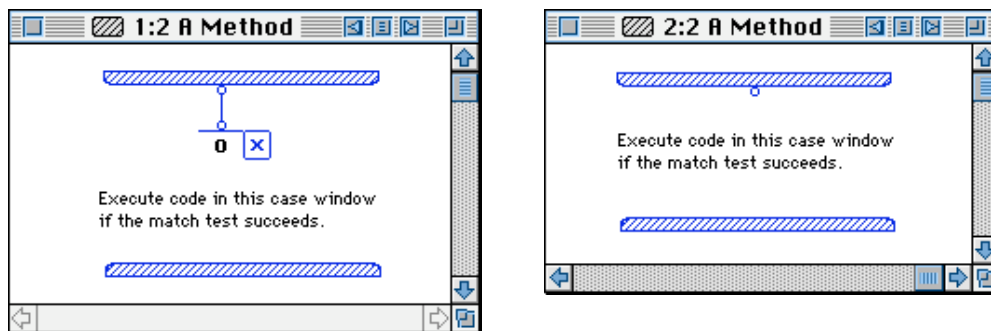


Figure 2.9: A “match” logical test

Exactly *what* is done following the *match* depends upon the type of *control* present on the right side of the match icon. In the above match, the “X” means “do something else if the test *FAILS*.” If the test *succeeds*, the remaining code in this code window continues to execute. If the test has *failed*, we must provide the “something else” -- the other code -- to be executed.

How do we provide the other code? In Prograph, the window in which dataflow code is contained is called a *case window*. Now we’ll see why. A *case* is a section of code that is executed when a test has been made. All methods by default have *one* case window -- even if *no* match tests are made in the method, the code in this case window is executed.

If a match test is made, *other* case windows must be provided to handle all of the possible outcomes of the match. In the case of our above match, we need two case windows -- one with code to execute if the test succeeds and one to execute if it fails (see Figure 2.10). In Prograph, the two alternative courses of program flow resulting from the match test are easily distinguished visually because they are housed within two individual dataflow windows. In textual languages like C++, the only thing that distinguishes two alternative program flows are two blocks of text following the “if” statement. As these blocks get bigger, their interrelationship gets even harder to determine.



```
void
AMethod( short num )
{
    if (num != 0) {
        // Execute case 2
    }
}
```

```

    }
    else {
        // Execute case 1
    }
}

```

Figure 2.10: Case windows for method containing a match and their C++ language equivalent

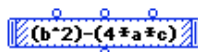
Notice the numbers in the titles of the two case windows preceding the method name. The second number, following the colon, indicates how many case windows this method has. The first number, preceding the colon, denotes the number of this case window. Our match will continue to execute the code in the default method case window (case window 1 out of 2) if the test succeeds, and will execute the code in the method's other case window (case window 2 out of 2) if the test fails. In this example, the program will execute different sections of code depending upon if the number input into the method is equal to zero or not.

As hinted above, it is the particular *control* attached to the match icon that indicates what is done next. We can test for either *success* or *failure* of a test -- that is, whether the outcome of the match is *true* or *false*. We can either execute new code following the test, continue executing the present code, execute a section of code and then stop, or even exit the section of code immediately. We'll discuss matches in more detail in Chapter 4, when we look at logical tests and program flow control, as well as Chapter 5, which covers loops.

Calculations with Evaluations

Last, but not least, is an operation that is very handy for carrying out small calculations when you don't want to write a completely new method to do so. This element is the *evaluation*. It is a small method-like symbol. However, unlike methods, it does not have its own code window. Its sole purpose is to hold a single mathematical *equation* to be evaluated. The example below, taken from the Prograph Reference manual, shows the first step of the calculation needed to solve a quadratic equation. It takes three inputs numbers at its three root nodes -- a, b and c -- and calculates the value b^2-4ac , which it returns as output from its terminal node.

A C++ language analog of a Prograph *evaluation* could be accomplished in one of two ways: either you could write a new function, incurring the cost of a function call, or you could implement the equation as a *macro* or *inline function definition* (see Figure 2.11). In either case, the function code is defined elsewhere in the program. The difference in Prograph is that the evaluation gives you a visual indication of what calculation the code is doing right there where it's used.



$$(b^2)-(4*a*c)$$

```

#define Quadratic(a,b,c) ((b*b)-(4*a*c))
    or
inline double
Quadratic( double a,double b,double c)
{
    return( (b*b)-(4*a*c) );
}

```

Figure 2.11: Calculation with a Prograph evaluation shown with its C++ language analog

Evaluations may take up to 26 inputs, one for each letter of the alphabet. Each one is named *sequentially* by each letter of the alphabet in turn. While this worked out nicely for our variables a, b and c, which happen to be the first three letters of the alphabet, it can be a problem if you want to name the variables x or y in the evaluation's equation -- for example, if you wanted to calculate x^2+y^2 to match the symbols used in an equation that you use regularly. To use the names x and y in the evaluation, you'd have to add 25 *root nodes* to the evaluation and enter the numbers in the two rightmost nodes. This would be very confusing. We suggest instead that you still use the variable names a or b in the evaluation equation, then comment the evaluation to show that x is equivalent to a, and y is equivalent to b.

Data Types - Constants, Variables and Persistents

Data in a Prograph program comes in three forms -- constants, variables and persistents. Each is used for different purposes, and each may hold one of several types of data values.

Constants -- data types whose values do not ever change -- are depicted as a horizontal line and terminal node with the constant's value above it. Constants may contain several types of data such as numbers, text or lists of items.


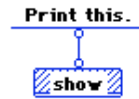

 const float fNumber = 45.0;

Figure 2.12: Prograph constant and its C++ language equivalent

Constants are used most in Prograph for data that is immediately fed into a primitive. For example, when we want to display a particular fixed text message with the `show` primitive, we can provide a text constant to it, as shown in Figure 2.13. In this case, we pass the message "Print this." to be printed immediately by the `show` primitive, but we do not need to store it for later use. Once the constant is used by `show`, it can be removed from memory.



```
cout << "Print this."
      << "\n";
```

Figure 2.13: Using a Prograph constant and a C constant

Variables of course are data types whose value *can* change. For example, when we enter a single number along a datalink from one operation into another, for example, a “-” primitive, we are supplying a *variable* to the “-” primitive whose value will be negated and returned to us for reuse. As another example, the *not* primitive *changes* a data value of a Boolean (true/false) variable from “*False*” to “*True*,” and vice-versa. Unlike in the C++ programming language, where variables *must* be declared and named, most variables are not given an iconic symbol in Prograph. They are simply passed along datalinks as “anonymous” data without their own label or symbol. There are two exceptions to this rule -- *persistents*, which we’ll discuss now, and *objects*, which we’ll spend a good number of pages discussing later in the book -- which are forms of named variables.

The typical variable exists *only for the life of your program* -- when your program is running. After you exit the program, the memory storage used to hold the value of a variable is freed up again, and its data value is lost. The next time you run the program, the variable must be given a value all over again. But there are times when you might want to store the last value of a variable and retain it for the *next* time you run the program.

This can actually be done with Prograph. A special data type called a *persistent* holds data while you run your program, then saves the value to disk storage when the program exits. When you start the program again later, the value of the persistent is *reset* to its former value by reading it from disk. Within the program, the persistent is used just like a *global variable* would be in other computer languages -- it can be accessed at any point in the program. A similar mechanism is just now starting to be provided by some C++ language libraries that implement what is called *persistent objects*. These are simply C++ data types that are stored to disk after program execution for later recall. However, the C++ programming language itself does not provide such a construct.

Prograph programs are made up of individual *sections*, which are program modules saved in individual files. Each section is composed of three components -- classes, methods and persistents, depicted in the three-faced symbol for a section. You create or examine pre-existing persistents by selecting the persistent face of a given section’s icon (Figure 2.14).

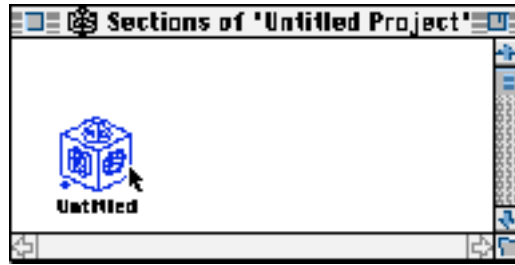


Figure 2.14: Opening the Persistents window

This action opens the section's Persistents window (see Figure 2.15).



Figure 2.15: The Persistents window

To set the type of data the persistent will hold, bring up the persistent's Value of Persistent dialog, seen in Figure 2.16.

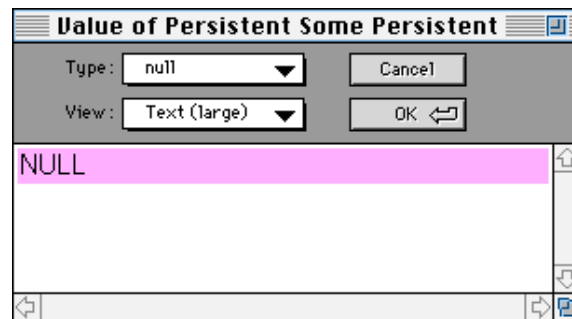


Figure 2.16: The Value of Persistent dialog

Select the persistent's data type from the *Type* pop-up menu in the dialog box. A default value of the persistent will appear in the bottom area of the dialog. You may change this value to any value that's acceptable for that particular data type. For example, as seen in Figure 2.17, if the data type is *real*, you can type in any floating-point number for the value of the persistent.

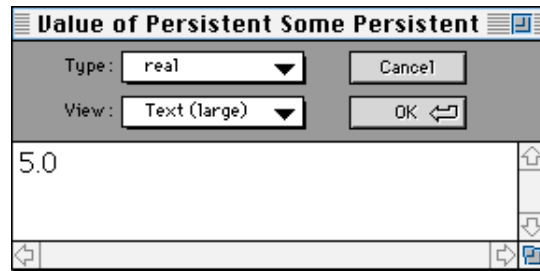


Figure 2.17: Setting the value of a persistent

Accessing the persistent's value from within a code window first involves creating a persistent icon in the window. Creating a persistent icon is similar to creating a method icon, but with one additional step. After you make and name a blank operation, highlight it and select the Persistent item from the Opers Menu. This will transform the blank operation into a persistent.

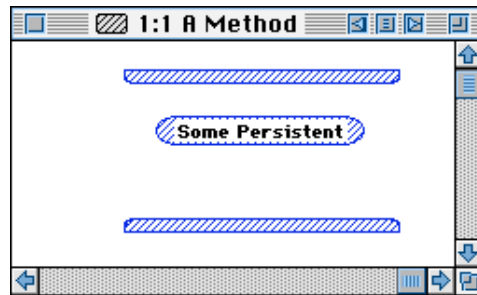


Figure 2.18: A persistent in a case window

The persistent icon has a characteristic shape -- an oval -- but it lacks any nodes. To feed a data value into the persistent or read a value back out of it, we must add our own nodes to it. To write data to a persistent, we create a terminal node on the top of the persistent; to read data from a persistent, we create a root node below its icon.

To read the value of the persistent, create a terminal node on its icon's bottom, then form a datalink from this node to the method to which you want to feed its data. In Figure 2.19, we display the value of Some Persistent by feeding its value, output from a root node, to the show primitive.

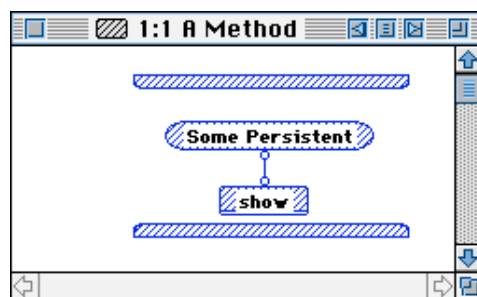
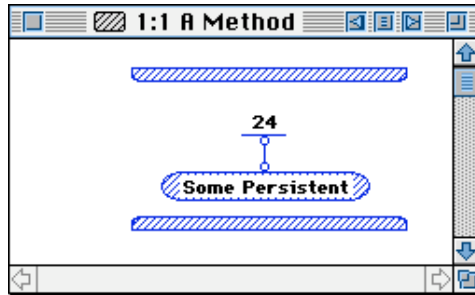


Figure 2.19: Reading the value of a persistent

Changing a persistent's value is done by adding a terminal node to the persistent's top, then connecting a constant or method output (variable or constant) to the root node of the persistent to feed a data value into it. In Figure 2.20, we initialize the value of Some Persistent to 24.

**Figure 2.20: Writing a value to a persistent**

Data Formats

Now let's examine the types of data values that constants, variables and persistents may hold, as well as the special uses for each.

Boolean

A *Boolean* is a special data type that *can only hold the values TRUE or FALSE*. Boolean data types are useful for holding program settings such as "Should my program use color?" (a *yes/no* value) in a variable we may change, read, test with logical decision-making operations. Boolean operators are not a built-in operator in the C++ programming language, but are so easily constructed that they are usually defined in C++ library files as enumerated data types where 0 denotes false and 1 denotes true.

Real

A *real* (short for *real number*) is a *floating-point* number, equivalent to the C++ IEEE-standard double-precision float (or *double*), which may be expressed either as a decimal number (e.g., 645,383.21) or in scientific notation (e.g., 6.4538321E5, which is equal to 6.4538321×10^5). A real's value may be any number in the range from -1.1E4932 to +1.1E4932.

Integer

In Prograph, an *integer* is a 32-bit *signed* integer number (a C++ *long* integer) whose value may range from -2,147,483,648 to +2,147,483,647. The real data type is Prograph's default numerical type, and can contain values that are much larger than this

range, so why would you need to use integers at all? Integers are useful because they may be manipulated much faster than floating-point numbers. When you don't need the decimal precision of a real number, an integer is preferable to use. In addition, many operating system routines expect integer rather than floating-point numbers as inputs or outputs.

External

This data type is used to represent several varieties of data types not directly supported by Prograph. Data of this type include C or C++ language *structures* or Pascal *records* that may be passed to operating system routines. External data types access data defined in external C or Pascal code written with the optional *Prograph C Tools* or *Prograph Pascal Tools* available from *Prograph International*.

String

Strings are collections of printable characters that may contain up to 65,535 characters each. Strings are built-in data types in Prograph. The C++ language does not support strings directly, but rather implements them as an array of individual characters. While the proposed ANSI Standard C++ library supplies some functions for manipulating strings, this is usually left up to the programmer.

List

Lists are special *collections of other data types*, such as a group of numbers or a group of strings. In fact, you may even collect lists of lists. Lists are somewhat similar to C++ arrays, but not quite -- in fact, there is no true equivalent of an array in Prograph. For example, lists, unlike arrays, do not have to be homogeneous; that is, a single list can hold several types of data. Since lists are so powerful, we'll cover lists in detail as a separate topic in Chapter 6. Lists are not directly supported by C++, but are a built-in data type in Prograph.

Null

This is a special data type used by Prograph that *holds no value at all*. It is, in effect, simply a place-holder for Prograph. Its C++ counterpart is NULL, defined as 0 or a void pointer, depending upon the compiler used.

None

This data type is somewhat like a Null, but rather than simply saying that the data has no value, *None* is used by Prograph to indicate that *no data at all* exists along this datalink. Therefore, you can't save None in a list like you could with a Null.

Classes






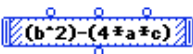
The final piece of the puzzle is provided by *classes*, which integrate *data*, and the *code* that manipulates the data, into a single programming element that can then be used by the programmer as if it were a built-in data type. The integration of code and data in classes can be seen in a class' two-part icon.

Classes are a way to build a library of new data types that can be reused in all of your programs (Prograph's *Application Builder Classes* are an example of such a library). Special properties of classes make them more powerful than just methods or data alone -- so much so that an entire mode of programming, *object-oriented programming (OOP)*, has been developed based upon classes. We'll start working with Prograph classes in Chapter 10.






Figure 2.21: A Class -- The basis of object-oriented programming



SUMMARY OF MAJOR PROGRAPH PROGRAMMING OPERATIONS

	Universal method operation	Operation that calls your methods
	Prograph primitive	Built-in Prograph method
	External method	System ROM method or method written in another language
	Local method	Local method
	Match	Logical test
	Evaluation	Calculation

SUMMARY OF PROGRAPH DATA TYPES

	Constant	Constant (unchanging) data
	Persistent	Permanent data storage
	Class	User-defined data type for object-oriented programming

SUMMARY OF METHOD INPUT AND OUTPUT

	Terminal node	Method input
	Root node	Method output