

6. Strings and Persistents

Overview

We're almost finished reviewing the background procedural programming material needed to start discussing object-oriented programming in Prograph. All that is left are some miscellaneous topics -- special data types that are built into the Prograph language, but are not typically found in other programming languages. In this chapter, we'll be examining how to use two new data types -- the text *string* and the *persistent*. Strings are special data types made especially for holding text, as opposed to the low-level collections of independent characters that other programming languages use to store text. Persistents are global variables that have special properties not seen in the global variables of other languages.

String Variables

Strings are simply text that is stored as a type of data. In Prograph, strings are a built-in data type (the "text" type) that is defined internally as *sequences of characters*. However, they are *used* as a complete entity of their own -- an *entire block of text or words* -- rather than as individual characters. This is immediately evident in both the code diagrams and in the Value dialog for strings, where strings are displayed as an entire block.

This is in sharp contrast to other languages, like C, where strings are nothing more than arrays of characters, and this is how they must be accessed. You can't automatically manipulate the string as a whole, but instead must write a loop to manipulate its individual characters. What a pain in the neck! Prograph was made to make programming simpler, and strings are no exception.

Until now, our use of strings has been limited for the most part to text *constants*. We've used text constants to prompt the user for input, to give directions, and to give greater meaning to the outputs of our programs. In the few programs where we've used string *variables*, that is, when we've had the user input text strings, we've done so without making any changes to the text. In other words, our only use of strings to date has been to make our programs more user-friendly. While this is not a bad thing to do, there are certainly many other uses for strings. Text may also be manipulated as a *variable* -- we may *concatenate* or join strings of text together, break down large strings of text into smaller *substrings*, or *test* strings or substrings for equality.

Concatenation

Some of our previous programs have required us to use the `show` primitive with a large number of input nodes in order to present a long string. It was easy at times to get these numerous nodes mixed up. The Prograph "`join`" primitive allows us to concatenate as many strings (including text constants) as we wish into one long string,

like calling the ANSI C library function *strcat()* repeatedly. The quotation marks surrounding the name of the “join” primitive remind us that this primitive is used only on strings (strings are usually delimited by quotation marks). Let’s start with a simple example.

Begin with a new Get Full Name program, section and universal method. Open the Get Full Name method window and introduce three *ask* commands which will input the user’s first name, middle initial and last name. For the *ask* used to enter the middle initial, inform the user that they do not have to enter a period after the initial (the program will insert it automatically). Ensure the proper order of execution of the three *ask* primitives by using *synchro links* (see Figure 6.1).

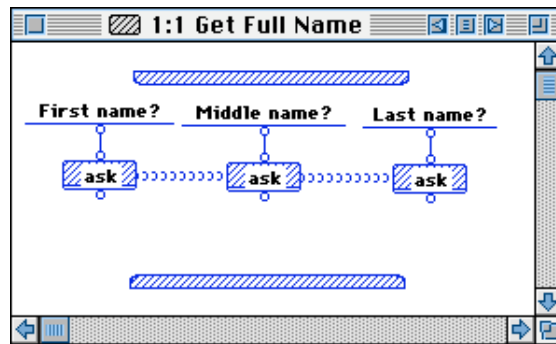


Figure 6.1: Initial code for the Get Full Name method

Create a “join” primitive under the three *ask* primitives. The “join” command will concatenate the three parts of the user’s name that were entered by the three separate *ask* commands. The visual nature of the code makes the joining of text together more obvious. The first input to the “join” primitive is the string containing the first name, output by the first *ask* primitive. The second input, to be appended to this string, is a text constant containing a *space* character to separate the text of the first name and the middle initial. A space character is denoted by a space between its quotation marks. Next, a third input, the string output by the second *ask* primitive containing the middle initial string, is concatenated to our string. The fourth input to the “join” primitive is a short text constant containing a period to follow the middle initial, plus a space to precede the last name. The final input to the “join” primitive is the last name, input using the third *ask* command.

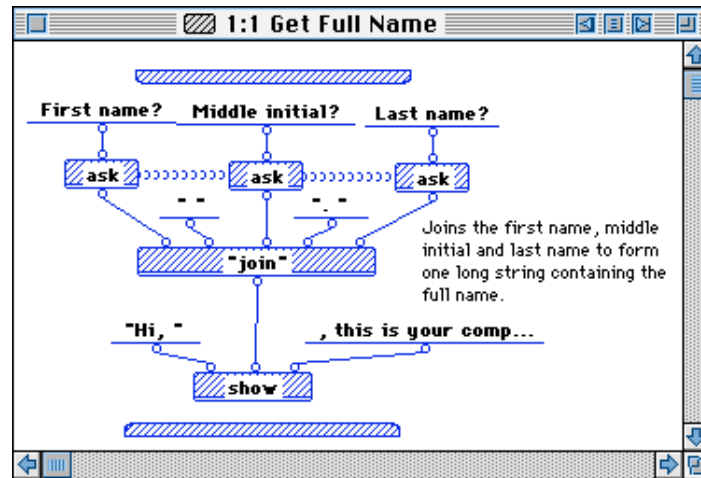


Figure 6.2: Completed code for the Get Full Name method

The final concatenated string contains the entire user name. This string is passed to a `show` primitive, which prints a greeting to the user, displaying the text “Hi, ”, then the user’s name, then another text constant containing “, this is your computer speaking!”.

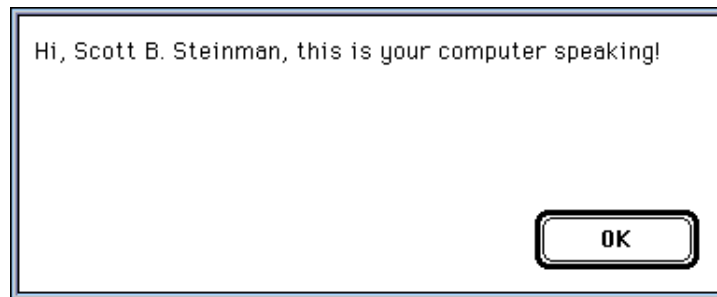


Figure 6.3: Output of the Get Full Name program

Notice that the `show` primitive itself can concatenate many inputs. Why then do we need to use the “`join`” primitive at all? If we had not done so, the `show` primitive would require *seven inputs* -- the two inputs needed for its text constants, plus *all five* of the inputs gave instead to the “`join`” command. This number of inputs to one primitive would be very unwieldy and confusing to the reader of this method’s code. The sheer number of inputs could, in turn, make the programmer more likely to make a mistake. By splitting the handling of the large number of strings between two primitives, we’ve made them easier to deal with.

Exercise 6.1:

Create a program to do the following: Ask the user for a subject (a noun), adjective, adverb and object (another noun). Construct a single string that will contain the sentence:

“The (adjective) (subject) jumped (adverb) over the (object).”

Have the program replace the four parts of speech in the sentence with those entered by the user, then display the new sentence.

Substrings

Now that we know how to *concatenate* or *join* strings together, what about the *opposite* problem -- accessing one smaller set or *substring* of characters contained within a larger string? Prograph provides three primitives to split off substrings from a larger string -- `prefix`, `middle` and `suffix`. They extract the beginning, middle and end sets of characters, respectively, from the large string input to them.

`Prefix` accepts a string input, and extracts from it the leftmost characters of the string. It outputs both this substring and the remaining rightmost characters of the original string (the *suffix* of the string). Just how many characters are extracted into the *prefix* is determined by a second input to the `prefix` primitive. `Suffix` acts in a similar manner to split off the suffix of a string, and outputs both the suffix substring and the remaining prefix of the string. Notice that both primitives do similar tasks. The difference is that `prefix` splits off *n* characters of *prefix*, and the `suffix` primitive splits off *n* characters of *suffix*. Each does not really care about how many characters remain in the original string when they're finished. As an example, if our input string is “Betty Boop”, a call to `prefix` with a request to split off 3 characters yields the substring “Bet” and the remaining suffix “ty Boop”. If we'd called `suffix` instead with the same request to split off 3 characters, we'd get the substring “oop” and the remaining prefix “Betty B”.

The `middle` primitive extracts the *n* characters requested from the *middle* of a string. It knows where in the string to begin its extraction by means of its *index* input, which tells it to start at a particular character. In other words, we tell the `middle` primitive to extract *n* characters starting at character *index*. The remaining characters of the original string are not returned. To reuse our example, a call to `middle` with a request to split off 3 characters (*n*) from the string “Betty Boop” starting at character 4 (*index*) would yield the substring “ty “ (with a space at the end).

We'll leave the application of these primitives as an exercise for the reader.

Exercise 6.2:

Ask the user to enter their address with a single `ask` primitive. Then store the house number, street name, city and state in separate persistents by separating the string entered by the user into its component substrings.

String Conversion

The “join” primitive used in Get Full Name has one major limitation: it only works with strings. But what if our program asked the user to input numbers? How could we create a large string (like a sentence) to contain these numbers? Luckily, Prograph provides yet another primitive to let us do this -- the `to-string` number-to-string conversion primitive.

Create a program called Numbers To Strings and its main method. Start entering the code diagram of this Numbers To Strings method as shown in Figure 6.4.

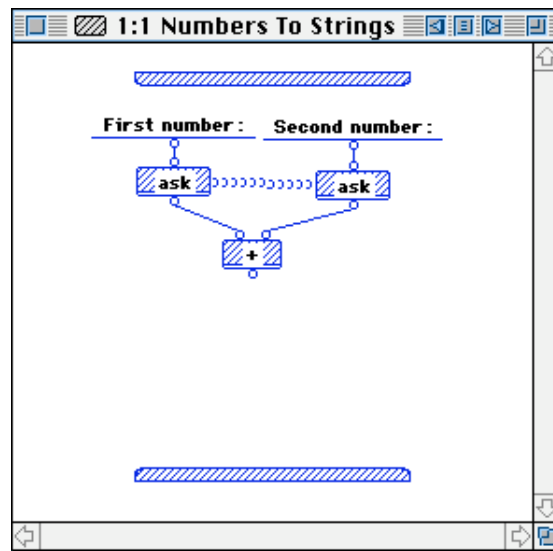
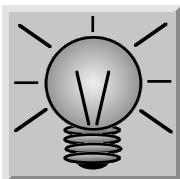


Figure 6.4: Initial code of the Numbers To Strings method

In this method, the user will enter *numbers* instead of text. We’ll add the two numbers together, then display the result in a sentence that reads “The sum of x and y is z.”, where *x*, *y* and *z* are three numbers. In this example, we’ll use the “join” primitive to create our output string for display. To accomplish this, we change the numeric data entered by the user or added by the + primitive into string data, using the `to-string` primitive (see Figure 6.5). The `to-string` primitive is very much like the ANSI C library `itoa()` and `ftoa()` functions, except that it is *polymorphic* -- it can output a *large variety of different data types*, including integers, reals, lists, and more. This sort of flexibility is only possible with Prograph.



A Hint...

The “join” primitive cannot handle leading or trailing spaces in the text strings it receives as inputs. To ensure that these spaces are printed properly, enclose all strings beginning or ending with spaces in quotation marks.

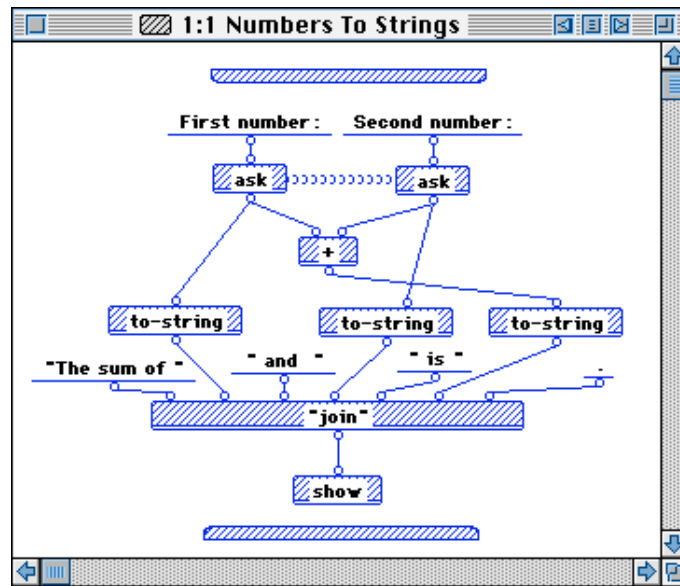


Figure 6.5: Completed Numbers To Strings method

We feed the output of each “join” primitive (a number in each one here), and the output of the + primitive (also a number), into three to-string commands. The string versions of each number can now be safely input into the join command to create the displayed message containing both text and numbers (Figure 6.6).

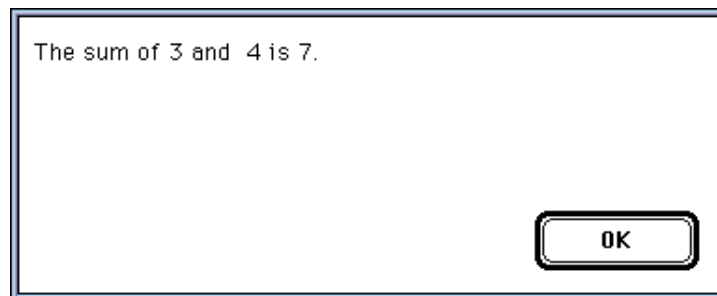


Figure 6.6: Output of the Numbers To Strings method

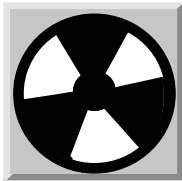
Persistents

Universal methods can be called over and over again anywhere in your program. But what about data? Can we create data elements that can be used and reused anywhere in our programs? Yes. In C++, these data elements would be called *global variables*. In Prograph, the equivalent data element is the *persistent*.

But persistents are much more than this! They don’t just hold their data while our program is running, as do simple global variables. Let’s show you what we mean...

In programming courses using C++, one of the first programs commonly written is a *checkbook-balancing program*. The program would begin with a starting balance for the checkbook account (stored in a global variable), and ask you to enter credits (deposits) and debits (withdrawals) from the account. The program would end by displaying a list of the credits and debits, and the final (updated) balance. Without fail, a student will ask how can you get the computer to “remember” the account balance so that if you run the program again in a month, you wouldn’t have to re-enter the account balance to use as a starting value. In C++, the solution is not easy -- the contents of the computer memory (including the memory location holding the global variable) are wiped out when the computer is turned off! You must store the balance onto a disk.

In Prograph, we can save the balance without explicitly storing it to disk ourselves. We do this with *persistents*. Persistents are *automatically saved to disk for you* when the program ends. The next time the program starts, the *previous value of the persistent will be read back from disk* to be reused. In this aspect persistents are similar in function to a recent development in C++ called *persistent objects*. The difference is that persistent objects are not built into C++ -- it requires extra code libraries, if you can find them. In Prograph, persistence is built into the language.



Warning!

The automatic saving of a persistent’s value is accomplished by modifying the storage of the Prograph program itself on disk. While this is easily accomplished when running a Prograph program within the interpreter, compiled programs pose a problem. We can’t go around modifying compiled program code each time we want to save a persistent’s value. In compiled programs, we use the save and load primitives to explicitly save the primitive’s value to disk then restore it the next time the program is run. What this means is that a Prograph persistent is truly and automatically persistent only *within the Prograph environment*; in *compiled* programs, they are simply the same as global variables unless we save and restore them ourselves.

We’ll now write a program that will store a *password* in a persistent. The user must enter the proper value of the persistent in order to be allowed to run the rest of the program. If the wrong value is entered, the program ends. It’s obvious that we don’t want to lose the value of the password each time we stop running the program, so we’ll keep it permanently stored with the program in a *persistent* for future reference.

Create a new program, section and method named Password. Open the Password method window and enter the code of Figure 6.7.



Figure 6.7: Reading the value of a persistent in the Password method

The persistent is created by adding a blank (unnamed) operation, then highlighting it and selecting the Persistent menu item. This changes the blank icon to a *persistent* icon. Name the persistent Password. It will store the current value of the password that the user needs to know to run the rest of the program. To pass the value of Password to the Get Password method, create a root node on the Password persistent icon, then connect it with a link to a new terminal node on the Get Password method icon.

What is the current value of Password? Open the Password persistent's Value dialog (see Figure 6.8). The value dialog tells you the type of data stored in a persistent and the value of that data. The Value dialog has two displays in it. On the top is a pop-up menu containing *possible data types* for the Password persistent. The current data type is displayed in the pop-up menu. A second pop-up menu allows you to choose the size of the font used in the dialog. On the bottom is the *value* of the currently-selected data. By default, the persistent is of type *null* (no particular data type) and contains the value *NULL*.

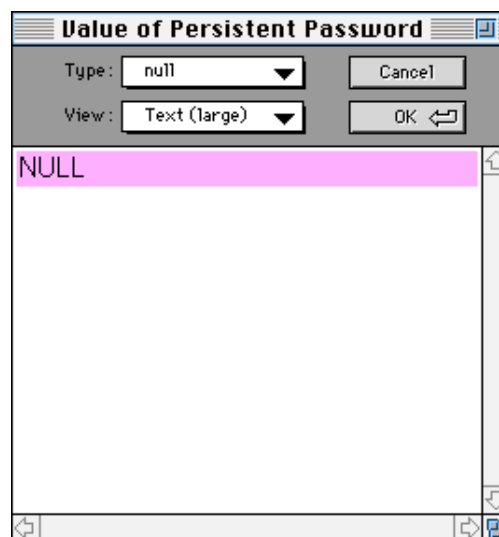


Figure 6.8: Viewing the value of a persistent with the Value of Persistent dialog

Let's change the Password persistent so that it will hold a *string*, then set the value of the string to "ABCD". This string will be our default password, which we'll change later. Select the *string* item on the pop-up menu. The content of Password is now a string type, but it is an *empty* string, as shown by its value ("") on the bottom of the dialog. Edit the string's value to read "ABCD", as shown in Figure 6.9, then click the OK button to close the dialog. Our Password persistent now contains the string "ABCD".

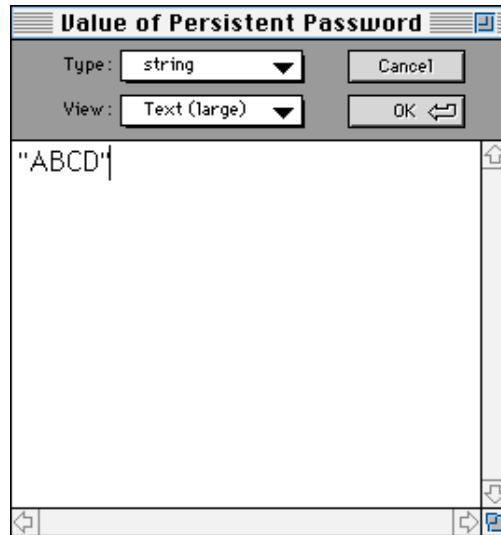


Figure 6.9: Entering a new value for a persistent with the Value of Persistent dialog

The Password method now will read the current value of the Password persistent (now "ABCD"), and call the Get Password method to ask the user to enter the password's value.

Now let's write the Get Password method. This method, shown in Figures 6.10-6.11, receives the current value of the Password persistent and asks the user to enter what they think the password is. The *match* test in this method checks to see if these two strings match -- that is, whether or not the user typed in the correct password.

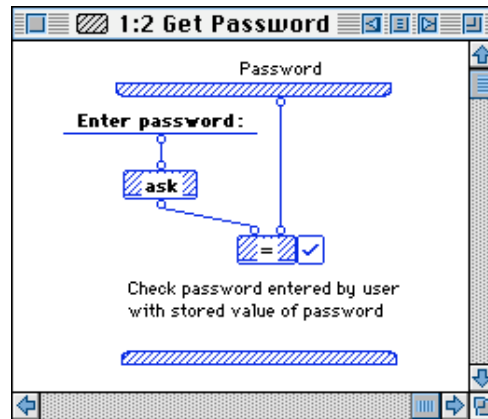


Figure 6.10: First case of the Get Password method

If the two strings don't match, that is, the user's guess for the password is incorrect, this method, and the program, end. If the user is *correct*, a second case for the Get Password method is entered which calls a Main Menu method that presents the *main menu* of program actions that the user may select.

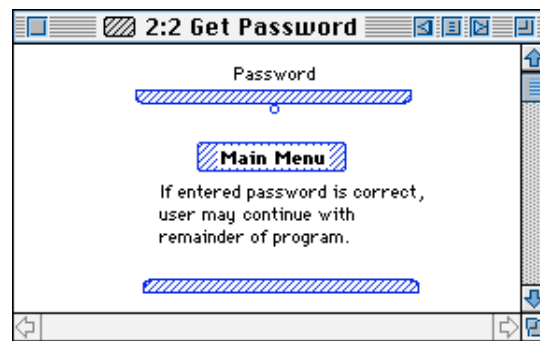


Figure 6.11: Second case of the Get Password method

The Main Menu method (Figure 6.12) presents the user with two possible choices of what to do next: continue to run the rest of the program, or reset the current password to a new value. Note that the user can't do either without having entered the correct value of the current password -- this code wouldn't have been reached at all. The user's selection of what to do is accomplished with the `answer-v` primitive, which passes the name of the method to be executed to an *inject* node. If you don't remember how to use *inject*, review the Choose Trig Function program on page 65.

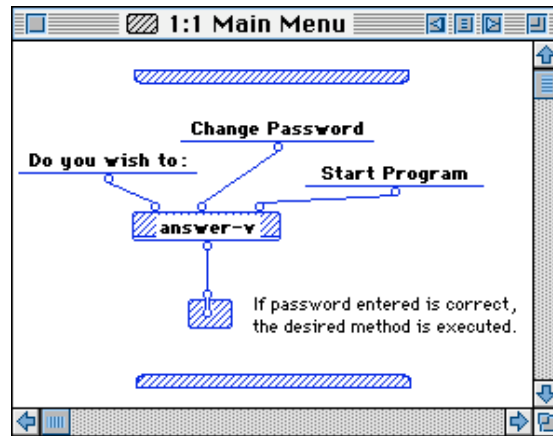
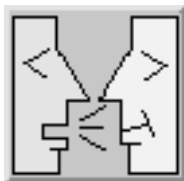


Figure 6.12: The Main Menu method

If the user wants to change the current value of the password stored in the Password persistent, the Change Password method is entered (Figure 6.13). This method replaces the current value of the Password persistent with a new value entered by the user.



By The Way...

Notice how easy it is to get or set the value of a persistent. To get its value, give it a terminal node on the bottom and send its value to a method with a link. To set its value, create a root node on its top and send it a value from a constant or a method.

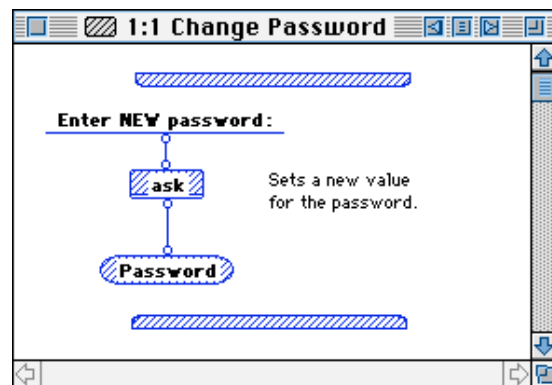


Figure 6.13: The Change Password method

If the user instead decides to run the rest of the program, the Start Program method is called (see Figure 6.14). In an actual program of your own design, this method would call all of the important code in your program, and do whatever it is you want the program to do. In this example, however, we'll just display a message that tells you that the Start Program method was executed.

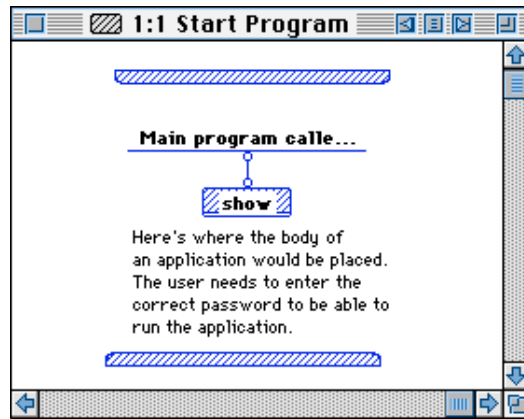


Figure 6.14: The Start Program method

Let's run the program. The program asks you to enter a password. Our initial password value had been "ABCD". Let's enter this value now so we can see how the rest of the program works (Figure 6.15).

A dialog box with a title bar. Inside, the text "Enter password:" is followed by a text input field containing the characters "ABCD". At the bottom right of the dialog is a button labeled "OK".

Figure 6.15: Prompt to enter the password

Once the correct password is entered, the main menu is presented, as shown in Figure 6.16.

A dialog box with a title bar. Inside, the text "Do you wish to:" is followed by two buttons. The top button is labeled "Change Password" and the bottom button is labeled "Start Program". The "Start Program" button is highlighted with a thicker border.

Figure 6.16: Dialog to select action

Select the “Change Password” item. A dialog will be presented (Figure 6.17) for you to replace the value of the password with a new value.

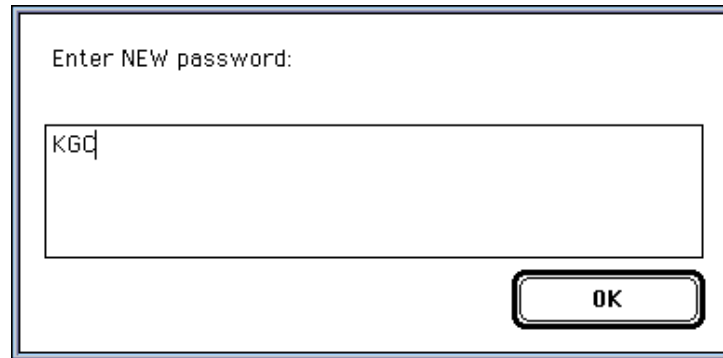
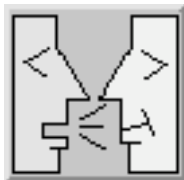
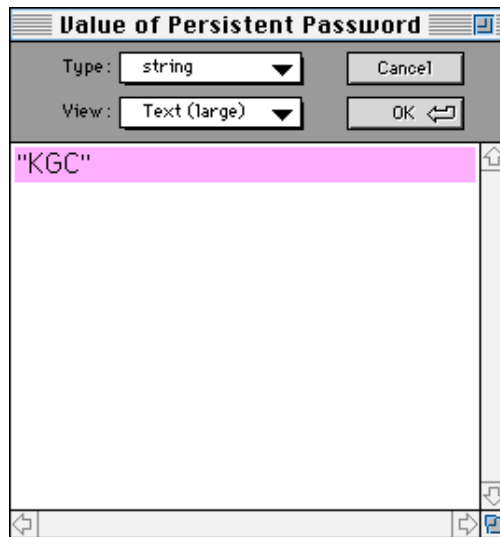
A rectangular dialog box with a thin border. Inside, the text "Enter NEW password:" is at the top left. Below it is a large rectangular text input field containing the characters "KGC". At the bottom right of the dialog is a button with the text "OK".

Figure 6.17: Prompt to enter a new password



By The Way...

You can confirm that this value is really stuffed into Password by checking the value of the Password persistent later:

A dialog box titled "Value of Persistent Password". It has a title bar with a standard icon. Below the title bar, there are two dropdown menus: "Type:" set to "string" and "View:" set to "Text (large)". To the right of these are "Cancel" and "OK" buttons. Below the dropdowns is a list view containing a single item, "KGC", which is highlighted in pink. The list view has a vertical scrollbar on the right and a horizontal scrollbar at the bottom.

If you’d chosen to start the program instead, you would have seen the message displayed in Figure 6.18:



Figure 6.18: Message indicating that body of program may be run

Exercise 6.3:

Write a savings account program which will present the user with the choice of withdrawing from a bank account or making deposits to the account, and keep track of the account balance. Use a persistent to store the running account balance. Assume a starting balance of \$250.00.

Exercise 6.4:

Write a lottery program which will calculate six random numbers between 1 and 49 for the selection of lottery entry numbers. The six numbers must all be different, so store each of the six numbers in a list stored in a persistent and check the value of each random number to the current contents of the list.

Summary

In this chapter, we covered two special Prograph data types:

- The ***string*** is used to hold sequences of characters, and may be used as either a text constant or a variable. It is the Prograph equivalent of the C++ array-based string. We have provided examples that demonstrate the use of strings in concatenation, splitting into substrings, and displaying numerical data.
- The ***persistent*** is Prograph's equivalent of the *global variables* of other programming languages. That is, the value of a persistent may be read or set anywhere in a program. But that's only part of what a persistent can do. Its value may be retained from one execution of the program to the next. This makes it especially well-suited for holding permanent but changing data.