

13. Applying OOP -- Creating Utility Classes

Overview

In this chapter, we will put the principles of object-oriented programming into practice. We will present several examples of object-oriented programming that may be used as building blocks for your own programs. Keep in mind that these examples may be far from complete -- they may not supply all of the functionality you might desire. However, the inclusion of fully-defined examples would require many more pages to display and explain than space permits. Our code is meant solely to serve illustrative purposes and provide some basic building blocks for your programs. Just as we have extended the range of available Prograph data types with our examples, we hope that you will extend the usefulness of our code.

Complex Numbers

As our first example of applied object-oriented programming, we will construct a new data type that may be used in your future Prograph programs as if it were included as a built-in data type of Prograph CPX. Our new data type will represent *complex numbers*, a mathematical construct built upon the concept of *imaginary numbers* -- the square root of -1. Although complex numbers have a theoretical background, they do have real-world applications, such as in electronic design, signal analysis and image processing. They are also fairly easy to implement, and therefore make a good initial example of creating abstract data types that can form building blocks for other programs.

Complex numbers have two components -- a “real” component (positive and negative numbers, or what we commonly think of as numbers in daily life) and an imaginary component. This is reflected in the two attributes of our **Complex** class, as pictured in Figure 13.1. This new class will be saved in a section of the same name.

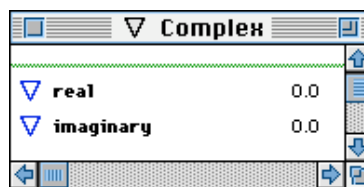


Figure 13.1: Real and imaginary component attributes of the Complex class

Setting the attributes of the **Complex** class or getting their current values requires writing our own class methods, since we would almost always want to work with both the real and imaginary components at the same time. The two methods for accessing the **real** and **imaginary** attributes are shown in Figures 13.2 and 13.3.

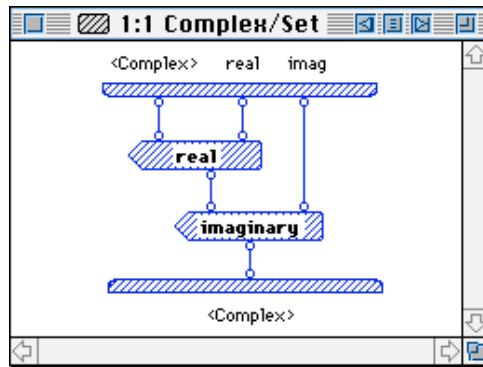


Figure 13.2: Set method of the Complex class

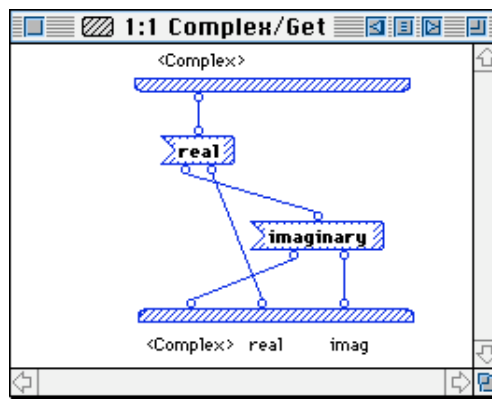


Figure 13.3: Get method of the Complex class

Mathematical operations on complex numbers take two flavors -- adding, subtracting, multiplying or dividing complex numbers with *other* complex numbers, or performing the same operations with *real numbers alone* (with no imaginary component). Let's take the second case first, since these operations are fairly simple.

Adding a complex number to a real number involves adding the real number to the complex number's real component (see Figure 13.4). Subtracting a real number from a complex number or multiplying or dividing complex numbers all follow the same logic, so we won't take up space showing you those methods.

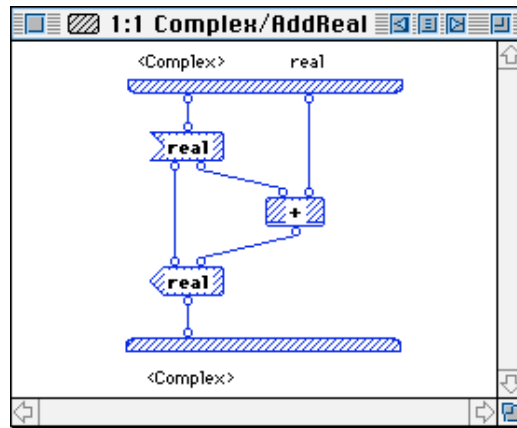


Figure 13.4: AddReal method of the Complex class

Adding two complex numbers together or subtracting one complex number from another are also fairly straightforward tasks. All that is required is adding or subtracting the **real** and **imaginary** components of **Complex**, respectively. The **Add** class method of the **Complex** class is shown in Figure 13.5. The **Subtract** method is the equivalent code with subtraction of the **real** and **imaginary** components.

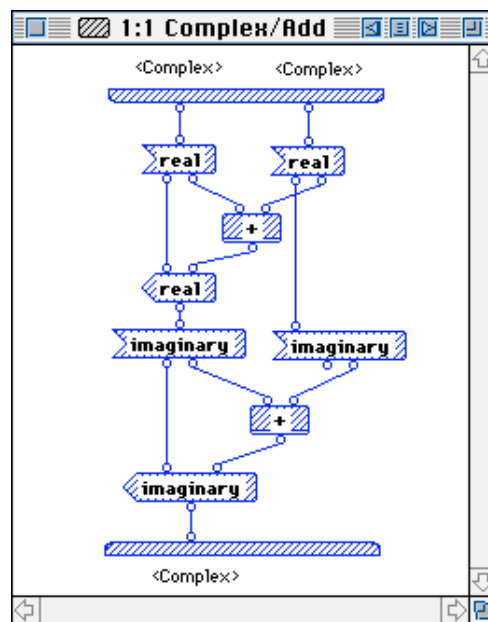


Figure 13.5: Add method of the Complex class

Multiplying two complex numbers is just a little bit trickier. The **real** component of the product **Complex** is computed by subtracting the product of the **imaginary** components of the multipliers from the product of the **real** components of the multipliers. To get the product **Complex**'s **imaginary** component, we add the product of the first multiplier's **real** component and the second multiplier's **imaginary** part to the product of the second multiplier's **real** component and the first multiplier's **imaginary** part. To put

this in equation form, $\text{Real}_{\text{product}} = (\text{Real}_1 * \text{Real}_2) - (\text{Imag}_1 * \text{Imag}_2)$, and $\text{Imag}_{\text{product}} = (\text{Real}_1 * \text{Imag}_2) + (\text{Imag}_1 * \text{Real}_2)$. Figure 13.6 shows the code for the **Multiply** class method.

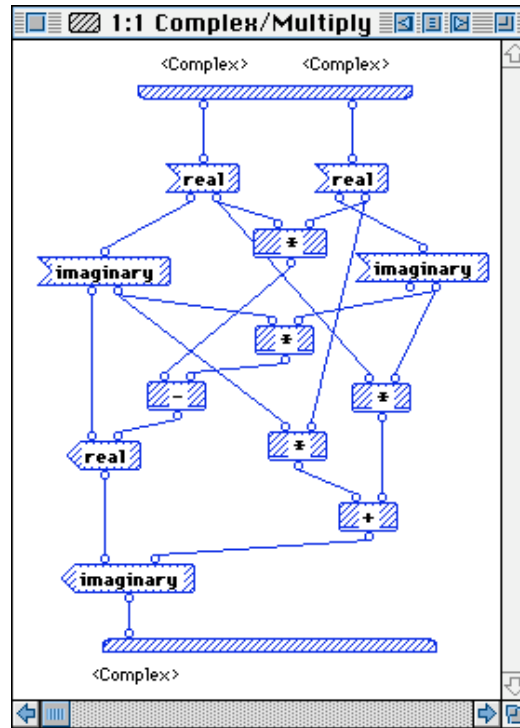


Figure 13.6: Multiply method of the Complex class

How do we divide two complex numbers? Division of complex numbers could be very complicated, but we can use a trick to make it simple. All we do is take the inverse of the divisor **Complex**, then multiply the dividend and inverted divisor complex numbers together (see Figure 13.7).

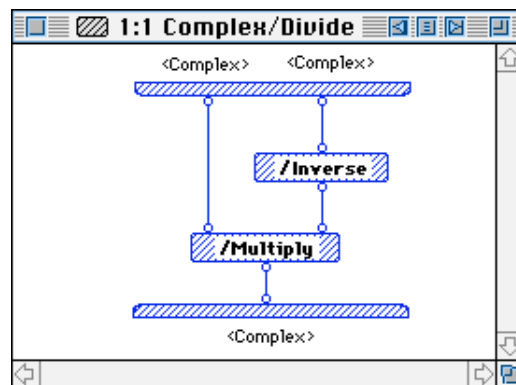


Figure 13.7: Divide method of the Complex class

Figure 13.8 shows the **Invert** class method of the **Complex** class. The inverse of a complex number takes the form $\text{Real}_{\text{inverse}} = \text{Real}_{\text{orig}} / (\text{Real}_{\text{orig}}^2 + \text{Imag}_{\text{orig}}^2)$ and $\text{Imag}_{\text{inverse}} = -\text{Imag}_{\text{orig}} / (\text{Real}_{\text{orig}}^2 + \text{Imag}_{\text{orig}}^2)$.

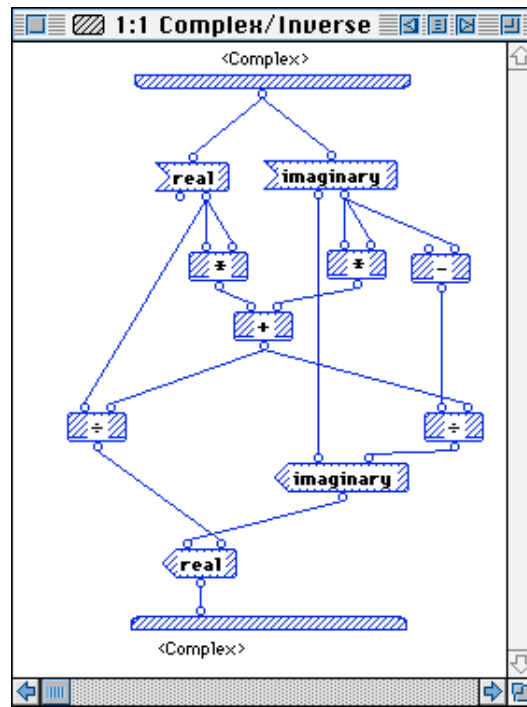


Figure 13.8: Inverse method of the Complex class

The elegance of the **Complex** class is that it makes complicated concepts easier to use. The details of the mathematics can be found in most math textbooks, and once written into the class methods, never have to be thought about again. Using complex numbers in a program becomes as easy as creating a **Complex** object, then sending messages to it to add, subtract, multiply or divide. We have made manipulating complex numbers in a Prograph program as easy as using integers or floating-point numbers.

Let's look at an example of the use of the **Complex** class. After saving the **Complex** section, create a new project with a new section called **Test Complex**. Add the **Complex** section to this project. In the **Test Complex** section, create a universal method called **Test Complex Class**. Complete its code as shown in Figure 13.9. This method creates two instances of the **Complex** class, sets their **real** and **imaginary** attributes, then adds them. The values of these attributes are read from the summed complex number and displayed. the output of the method is depicted in Figure 13.10.

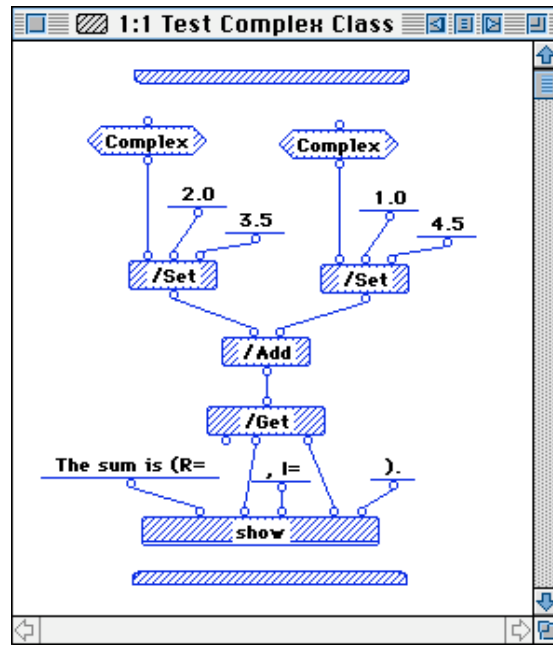


Figure 13.9: Test Complex Class universal method

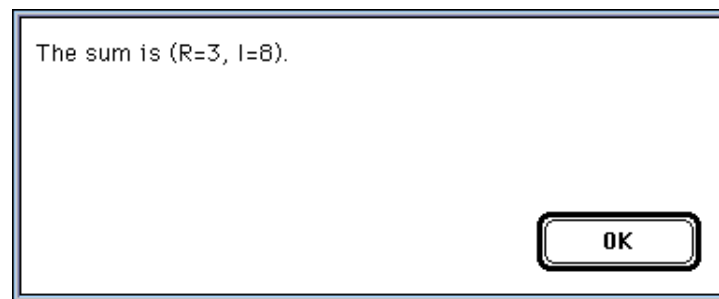


Figure 13.10: Output of the Test Complex Class universal method

Fractions

Another mathematical data type that may be encapsulated into a simple class is a *fraction*. We will design a class called **Fraction** that will simplify complicated fractions, add them together, subtract, multiply or divide them, or calculate their reciprocal.

The class requires only two attributes. One is the integer numerator of the **Fraction** and the other, also an integer, is its denominator (Figure 13.11).

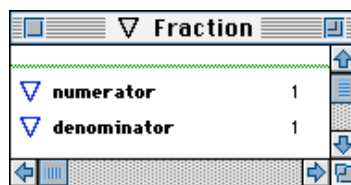
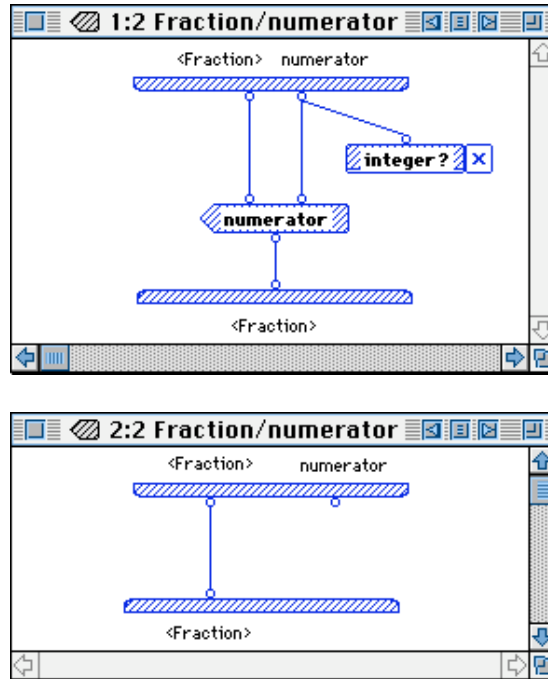


Figure 13.11: Attributes of the Fraction Class

Because the `numerator` and `denominator` attributes are integers, we should confirm that the values we wish to place in them are integers. We therefore write our own Set methods to automatically performs this test. The Set method for the `numerator` is shown in Figure 13.12. The Set method for `denominator` is similar.

**Figure 13.12: Custom Set method for the numerator attribute**

We also include a `Display` method, depicted in Figure 13.13, that puts the `Fraction` into a form that may be displayed on a computer screen, saved in a file or printed. It places into a string the `numerator`, followed by a "/" character, then the `denominator`.

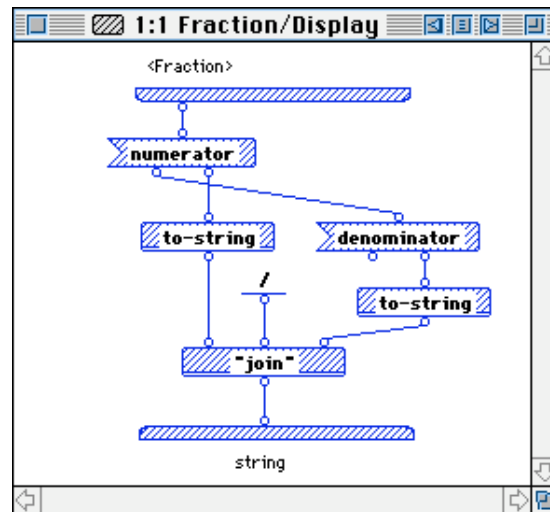


Figure 13.17: Display method of the Fraction class

The **Reciprocal** method calculates the reciprocal of a fraction by switching the values of the numerator and denominator of a Fraction. Its code is shown in Figure 13.18.

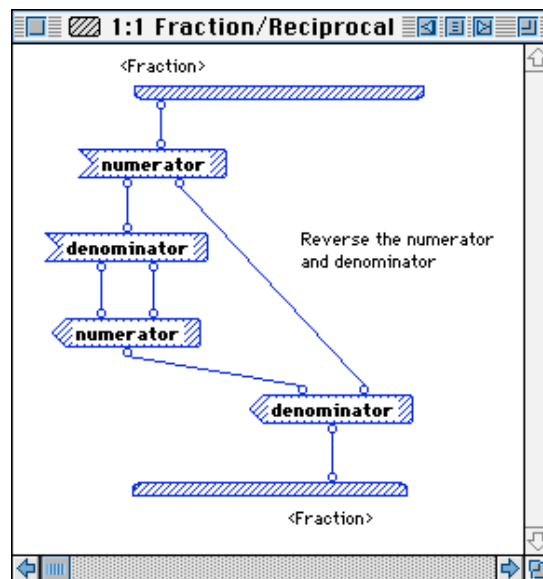


Figure 13.13: Display method of the Fraction class

All of the remaining methods of the Fraction class rely upon two utility class methods that simplify or *reduce* the fraction's value or allow mathematical operations on two fractions by calculating their *Lowest Common Denominator*. Let's look at the **Reduce** class method first.

The **Reduce** method is shown in Figure 13.14. It determines the largest number by which both the numerator and denominator are divisible via the reduce local method. It then divides each by that number. This has the effect of simplifying a fraction. For example, if the fraction is 3/12, the **Reduce** method will find that the largest number that both the numerator and denominator are divisible by is 3, and will divide each by that number. The fraction's value is therefore set to 1/4, which is a simpler way of stating 3/12.

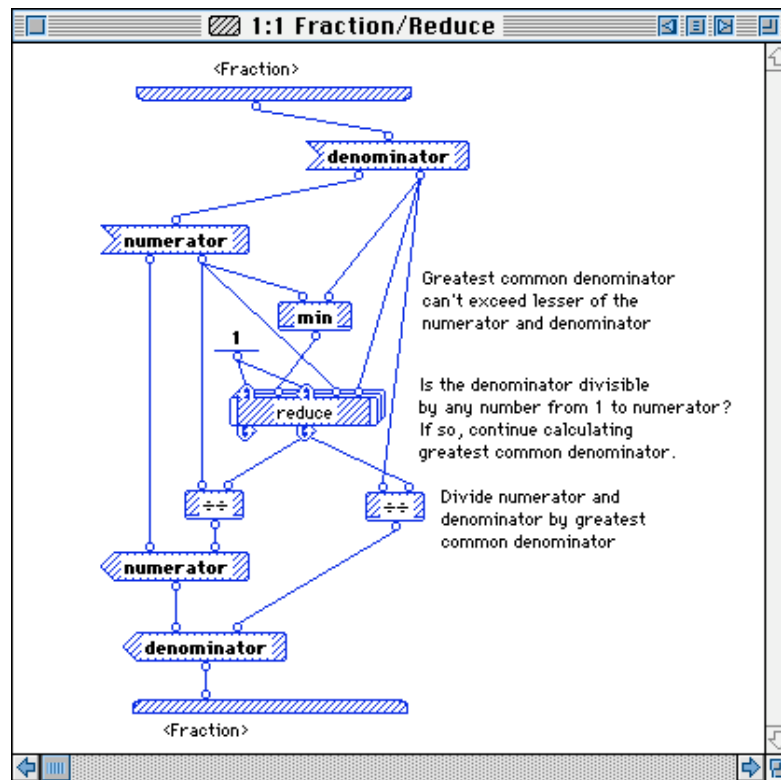


Figure 13.14: Reduce method of the Fraction class

The reduce local method is a loop that repeatedly calls a Get GCD local method. This local method (see Figure 13.15) checks if both numerator and denominator are integer-divisible by the loop counter (that is, integer division of each produces no remainder) using a logical *and* operation.

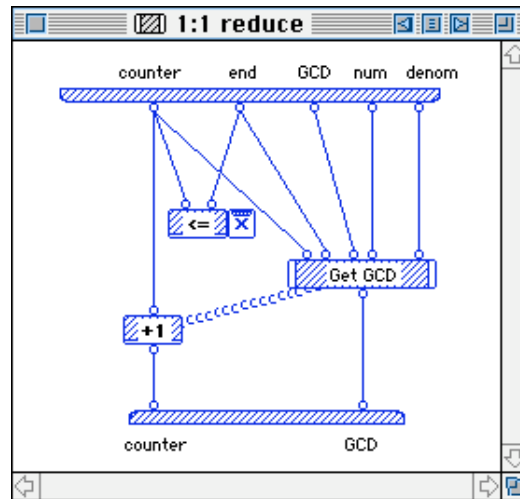


Figure 13.15: The reduce local method

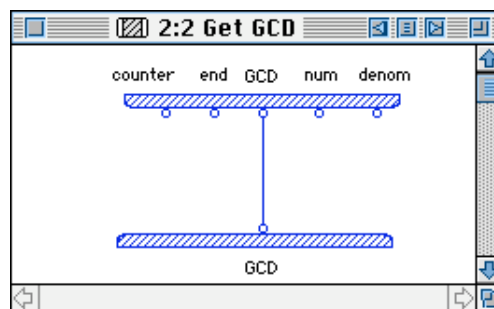
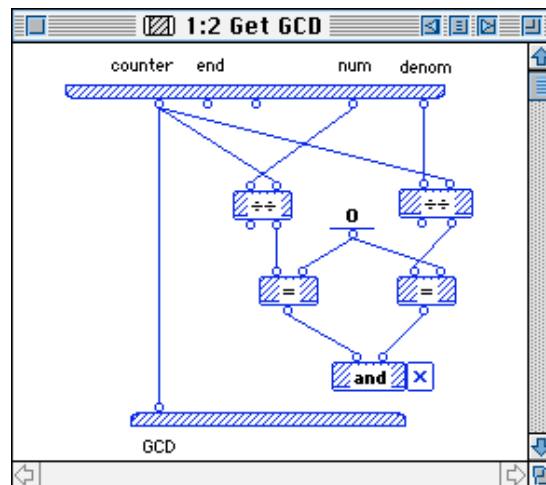


Figure 13.16: The Get GCD local method

The LCD method converts two Fractions so that they share a least common denominator. This is accomplished in two steps (see Figure 13.17). First, the denominators of each Fraction are multiplied together to form the new denominator of

each Fraction. Next, the numerator of one Fraction is multiplied by the denominator of the other to form each Fraction's new numerator.

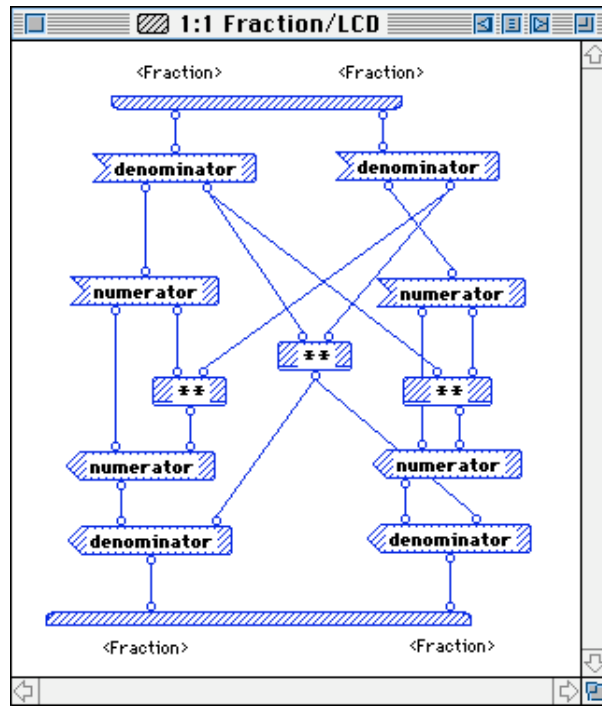


Figure 13.17: LCD method of the Fraction class

The GCD and LCD class methods form the common components of our remaining Fraction class methods. Let's start with the Equal? method, which determines if two Fractions are equal in value (Figure 13.18). The two are first reduced to their simplest form via the GCD method. Then the numerators and denominators are compared. If both are equal, the two Fractions are equivalent.

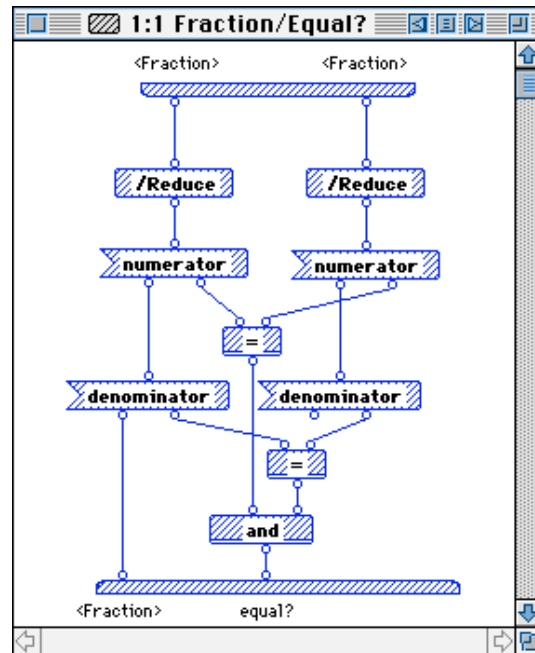


Figure 13.18: Equal? method of the Fraction class

The Add class method, shown in Figure 13.19, converts two fractions to share a common denominator by calling LCD, then adds their converted numerators together to form the summed Fraction's numerator. The Fraction is then reduced via the GCD method.

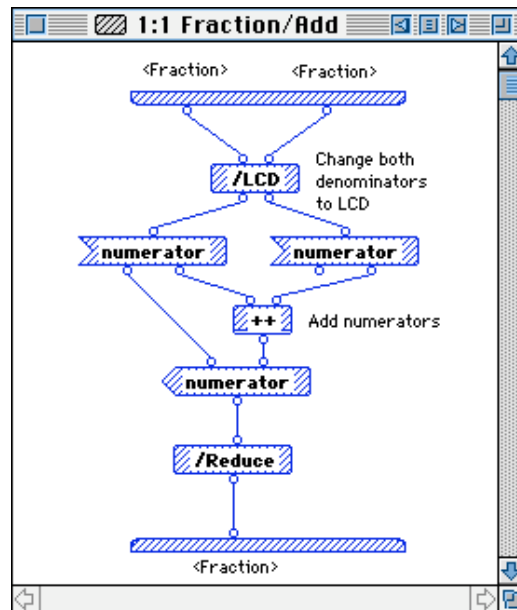


Figure 13.19: Add method of the Fraction class

The **Subtract** class method is almost identical to the **Add** class method. It performs all of the same steps as **Add** except that it subtracts one numerator from the other after converting them with LCD (Figure 13.20).

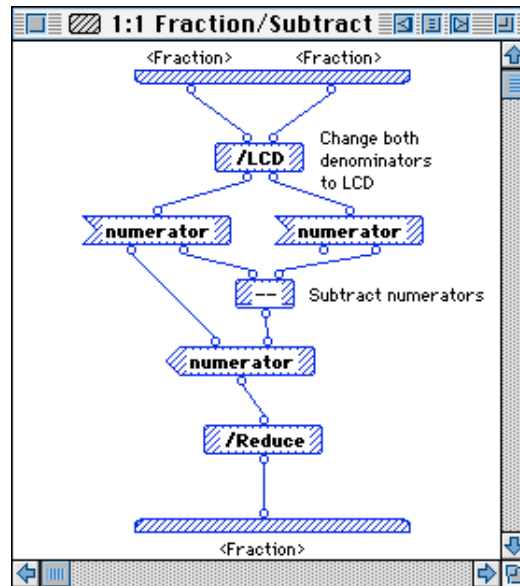


Figure 13.20: Subtract method of the Fraction class

The **Multiply** class method calculates the product of two **Fractions** by first multiplying their numerators to form a new numerator, then multiplying each **Fraction's** denominator to form a new denominator (see Figure 13.21).

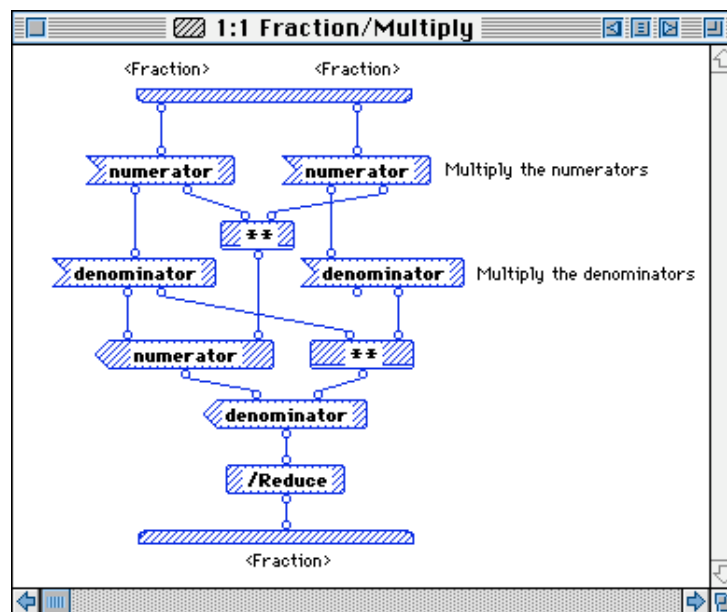


Figure 13.21: Multiply method of the Fraction class

The **Divide** class method (Figure 13.22) first converts the divisor **Fraction** to its reciprocal, then multiplies this by the dividend **Fraction**.

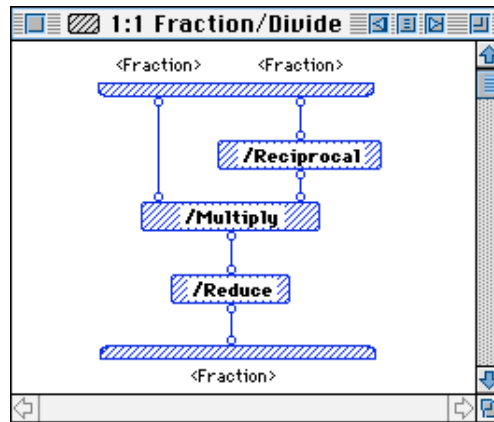


Figure 13.22: Divide method of the Fraction class

This completes the **Fraction** class. There is only one action that is missing from our class -- the ability to convert between decimal floating-point numbers and **Fractions**. We will leave the writing of these conversion methods as an exercise for the reader.

Stacks and Queues

Let's turn now to two classes that implement *stacks* and *queues*, two computer theory constructs familiar to most computer programmers which may be used in a wide variety of computer programs. In C++, object-oriented versions of stacks and queues are typically built upon arrays. Prograph does not have a built-in *array* data type, but we constructed one in the previous chapter. We could use the **Array** class to construct the new **Stack** and **Queue** classes, but in this case, a simple *list* will suffice rather than an **Array**. Building stacks and queues with lists simplifies our programming task considerably.

Starting with the **Stack** class, we see that its attributes consist of just a list to represent the stack's contents and an integer value to indicate which member of the list is at the current top of the stack (see Figure 13.23). We can see now why a list representation is preferable for the **Stack** class. Using a list will automatically provide us with an *unlimited* stack size, while even a resizable array would require constant resizing as the stack grew or shrank.

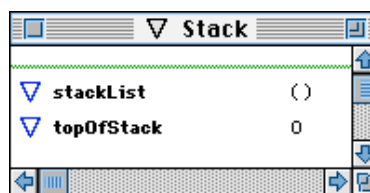
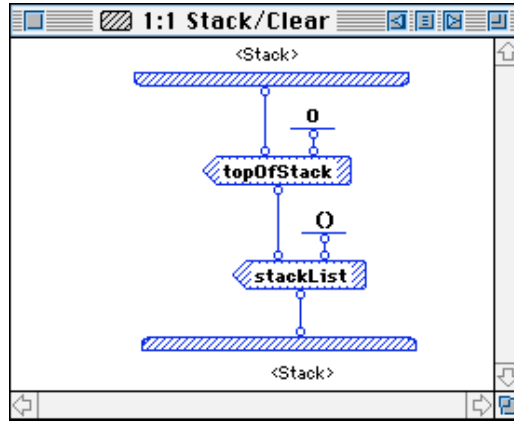
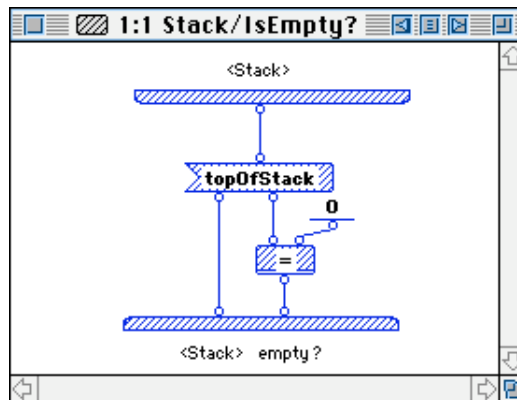


Figure 13.23: Attributes of the Stack class

Clearing a Stack involves emptying the contents of the `stackList` and resetting `topOfStack` to 0 by calling the `Clear` method (see Figure 13.24).

**Figure 13.24: Clear method of the Stack class**

As stated above, our stack can grow to any arbitrary size since the `stackList` is by definition resizable. We can keep adding items to the stack indefinitely. But what about *removing* items? When the stack is emptied, we'd like to check that we are not trying to remove items from an already empty stack. The `IsEmpty?` method, shown in Figure 13.25, allows us to test for this condition.

**Figure 13.25: IsEmpty? method of the Stack class**

Now let's move to the main actions of stacks -- *pushing* and *popping*, putting items onto and removing them from the stack. since the `Stack` class implements the stack with a list, all we need to do to push an item onto the top of the stack is attach an item to the end of the `stackList` and increment `topOfStack` by one. The `Push` method does both (see Figure 13.26).

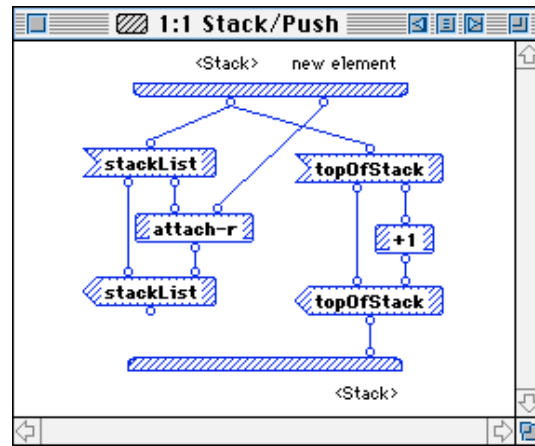
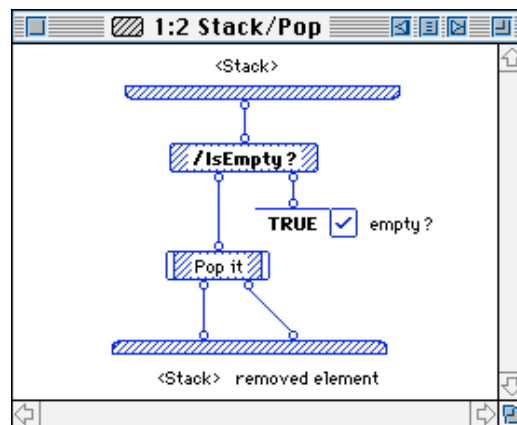


Figure 13.26: Push method of the Stack class

Popping an item off the stack is accomplished by the Pop method (Figure 13.27). This method first checks if the stack is already empty by calling the `IsEmpty?` method, and displays an error message if so. If the stack is not empty, the item at the top of the stack is removed in the opposite manner as it was pushed -- detaching the last element of `stackList` and decrementing `topOfStack` by one.



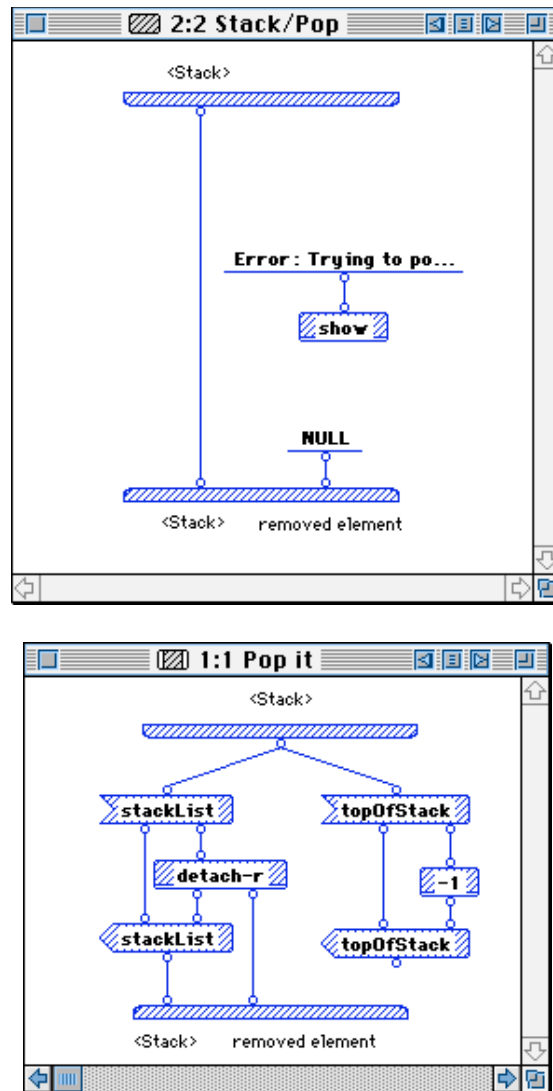


Figure 13.27: Pop method of the Stack class

Let's see how the **Stack** class performs. Our test stack method, depicted in Figure 13.28, pushes two integer numbers onto the stack. It then pops these values off the stack and displays them. A third pop operation is also attempted.

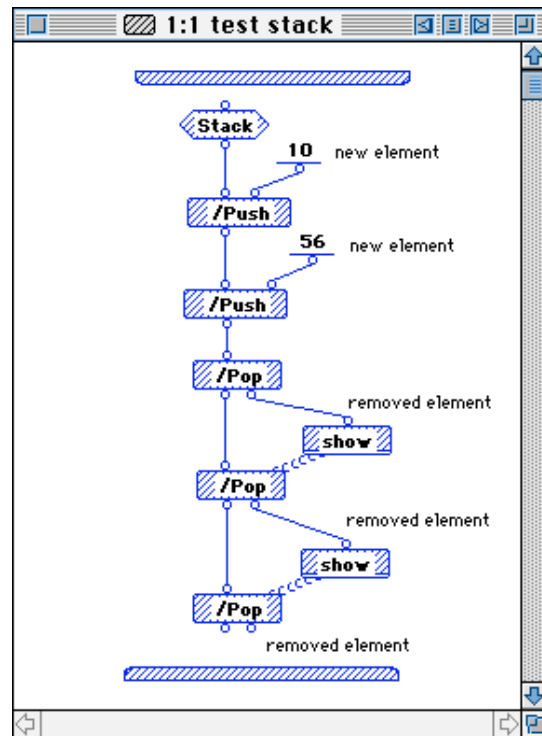
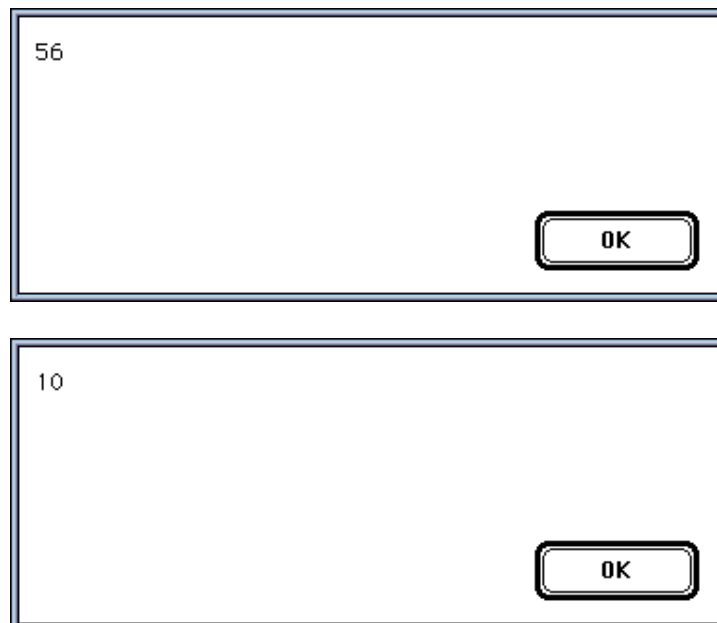


Figure 13.28: The test stack universal method

Note that the values pushed onto the stack are popped in the opposite order. This is the proper way that a stack should behave -- the last item in is the first item out (commonly abbreviated as LIFO). When the third pop operation is attempted, an error message is presented to the user that a pop was attempted on an empty stack. Figure 13.29 shows the output of the test stack universal method.



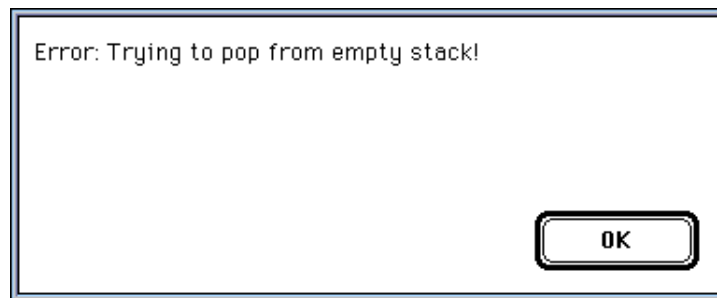


Figure 13.29: Output of the test stack universal method

The implementation of a queue is very similar to that of the **Stack** class. The sole difference in their behavior is that whereas a stack is last in - first out, the queue is first in - first out (or FIFO for short). The **Queue** class (Figure 13.30) only has one attribute, the **queueList**, since we don't need to maintain an index to the last item in the queue. It is always the last item of the list that forms the queue.

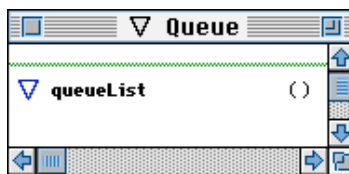


Figure 13.30: Attributes of the Queue class

The instance method, **Clear** method and **IsEmpty?** methods of **Queue** are identical to those of the **Stack** class, so we won't show them here. The **Push** and **Pop** methods, on the other hand, differ. **Push** simply adds a new element to the end of the **queueList** (see Figure 13.31).

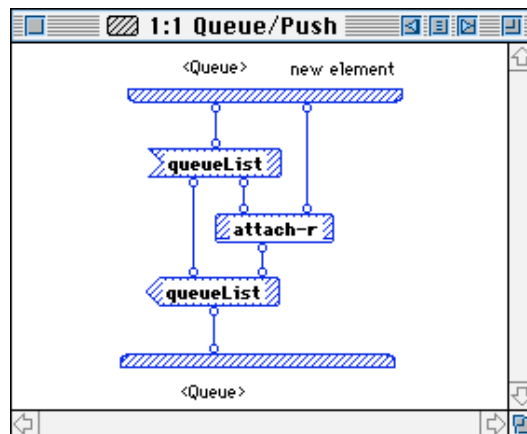


Figure 13.31: Push method of the Queue class

The **Pop** method, after checking that `queueList` is not empty, removes the *leftmost* element of the `queueList`, the *first* element that had been pushed onto the queue (see Figure 13.32).

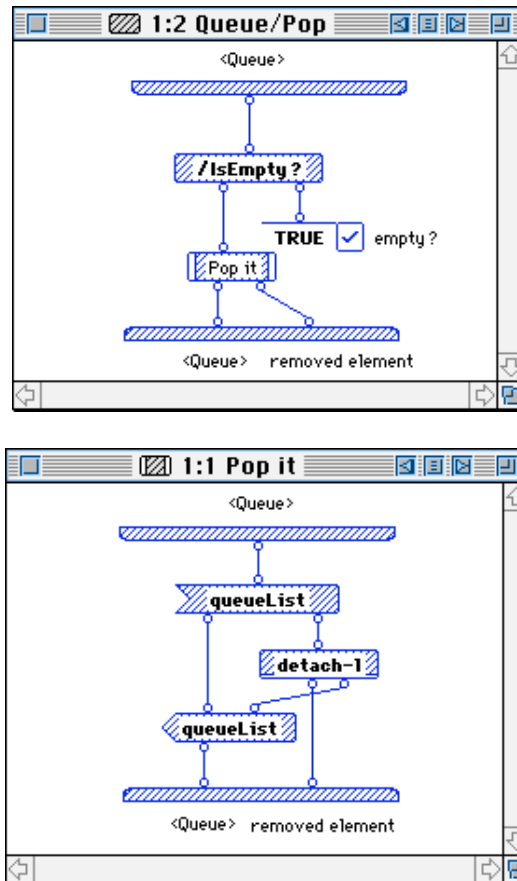


Figure 13.32: Pop method of the Queue class

The test queue method (Figure 13.33), which performs the same actions as the **Stack** class' test stack method, presents the popped numbers in the *same order* they were originally pushed onto the queue (remember, queues are FIFO). Figure 13.34 shows the output of the test queue method.

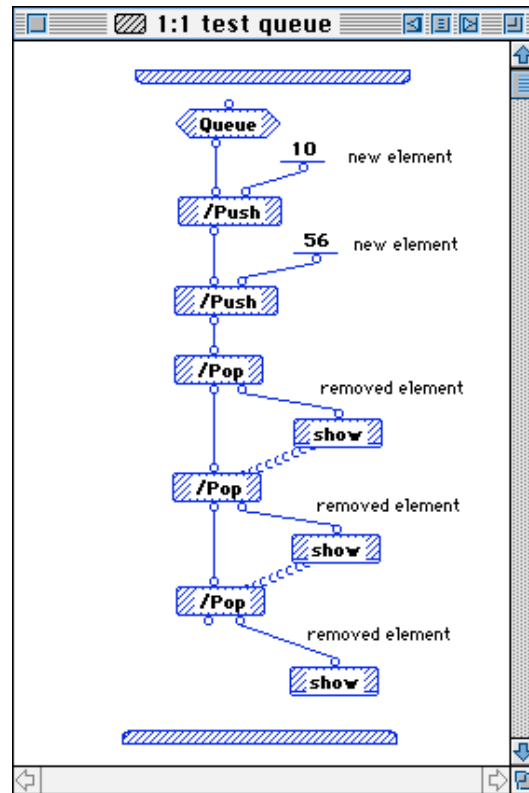


Figure 13.33: The test queue universal method

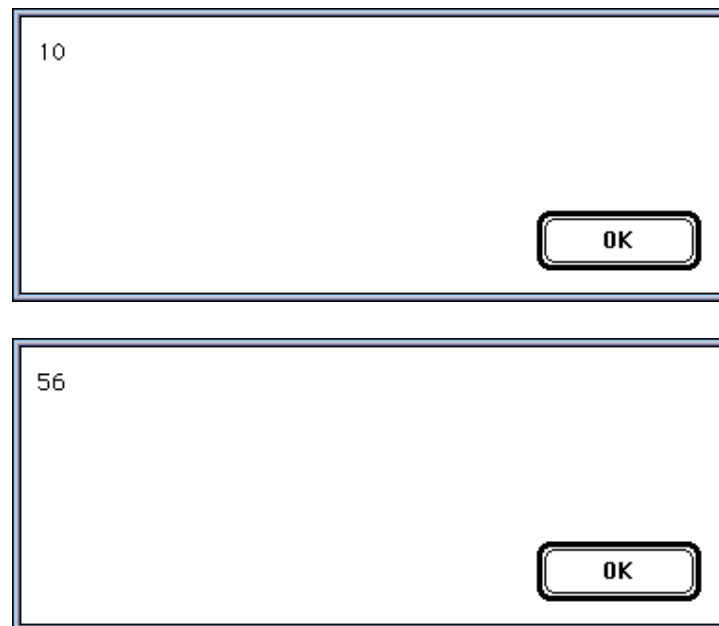


Figure 13.34: Output of the test queue universal method

Note that the `Stack` and `Queue` classes share much of their code. Shouldn't we have subclassed both from a common parent class containing a `list` attribute and the

instance, `Clear`, `IsEmpty?` and `Push` methods? Yes! Normally, we would. However, in this case, the two classes are so small that this point is moot. For illustrative purposes, we have kept the two classes separate.

Matrices

Our final utility class is another mathematical construct -- the *matrix*. Why are we discussing one more mathematical class? Matrices are arguably the most important and widely-used mathematical construct in computer programming. They are used not only in scientific programming, but also in image processing and two- and three-dimensional graphics. Their widespread use makes it a perfect candidate as a reusable utility class.

The **Matrix** class implements a special subset of matrices called *square* matrices -- matrices that have an equal number of rows and columns. Square matrices are the type of matrix used most commonly in computer programming and are much simpler to construct and manipulate. Rather than storing the square matrix directly as a nested two-dimensional list, we will build our **Matrix** class upon a set of utility classes that are related to the **Array** classes discussed in Chapter 11. The new classes are the **2-D Array**, or two-dimensional array, classes. These classes place a layer of abstraction between the matrix and the nested list used to store it. This “nesting” of one class inside another (composition) is also an example of one type of interclass communication. Objects created from the **Matrix** class must “talk to” the **2-D Array** objects embedded within them to access their matrix contents. While this may seem at first to unnecessarily add an extra level of indirection that could slow down matrix calculations, the use of the intermediate **2-D Array** classes greatly simplifies the programming of many matrix calculations.

The two-dimensional array classes are subclassed from a common **2-D Array** class. This class has three attributes shown in Figure 13.35. The `arrayList` holds the elements of the two-dimensional array in a nested list. The size of the nested list is defined by both the `rows` and `columns` attributes. `Rows` determines the number of sublists within `arrayList`, and `columns` defines the length of each sublist.

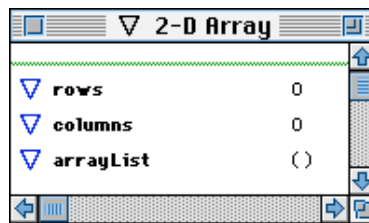


Figure 13.35: Attributes of the 2-D Array class

The `GetElement` and `SetElement` class methods are the most frequently called methods. `GetElement` (see Figure 13.36) checks that the requested array element row and column are within the bounds defined by the array’s `rows` and `columns` attributes,

then calls the `get-nth` primitive to read the appropriate item in the `arrayList`. The `SetElement` method (not shown) performs all of the same actions except that it calls `set-nth!` to replace the current value of the desired array element with a new value.

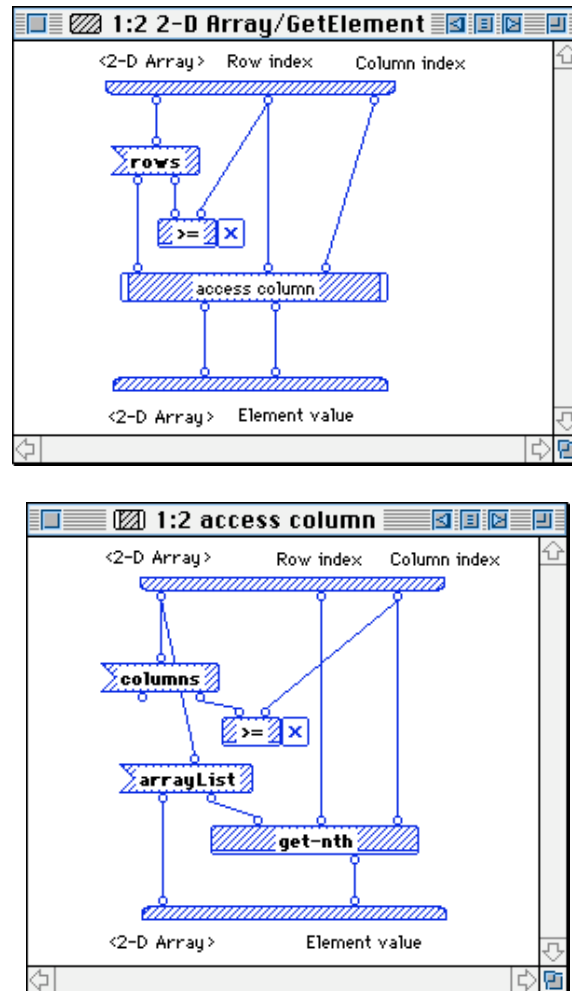


Figure 13.36: GetElement method of the 2-D Array class

The `Copy` class method creates a duplicate of a 2-D Array object. Rather than instantiate a new object, then use a loop to copy each element in turn, we use a short-cut (see Figure 13.37). Prograph CPX includes a `copy` primitive that performs a so-called “*deep copy*” of an object. In other words, it creates an exact byte-by-byte copy of an object, *including* nested objects (see the composition technique, discussed in Chapters 9 and 12).

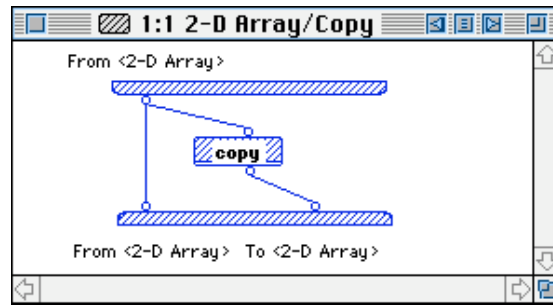
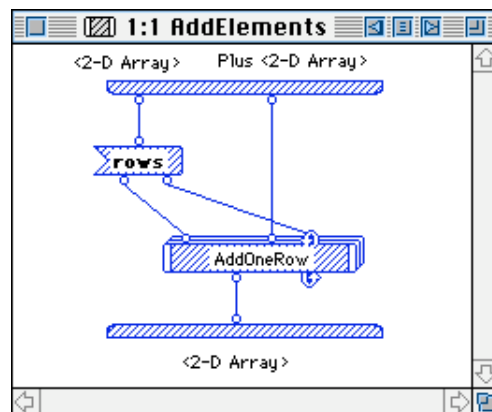
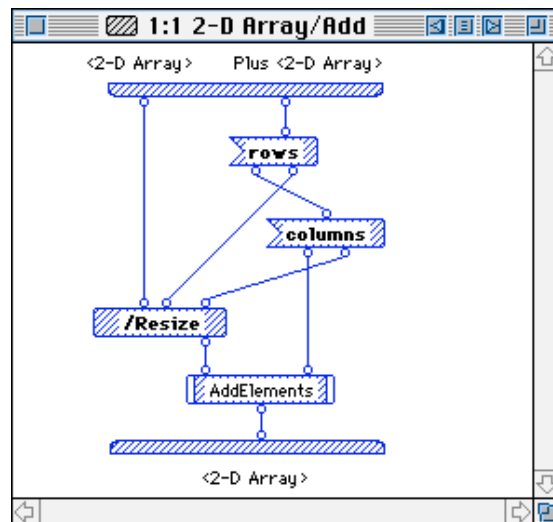


Figure 13.37: Copy method of the 2-D Array class

Adding arrays does require using a loop to access each element in turn. In the Add method (Figure 13.38), we first ensure that the array to be added to our original array is of the same size, resizing it if needed. We then enter a nested loop to add together each of their corresponding elements. The sums are stored in the original array.



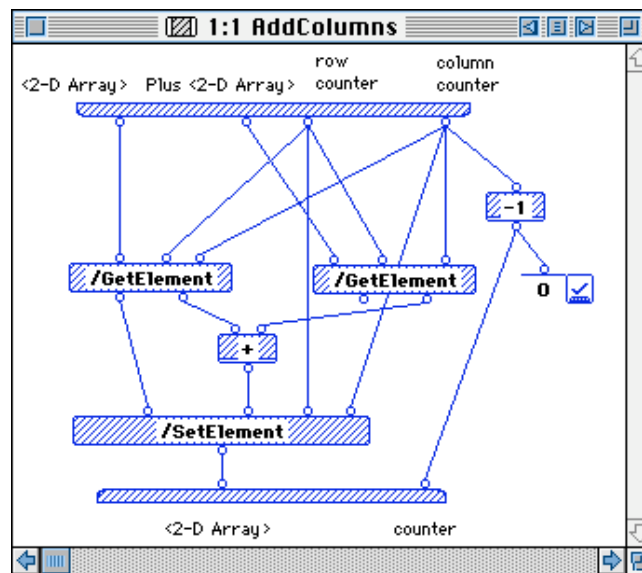
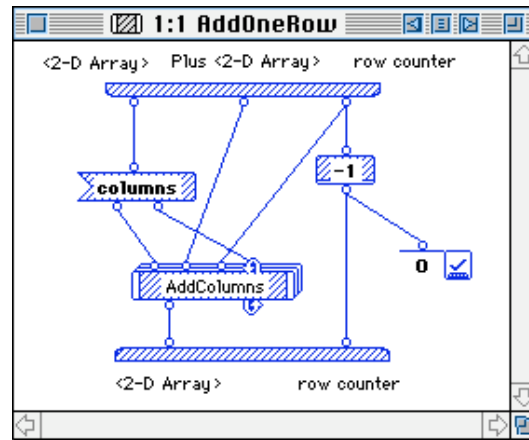


Figure 13.38: Add method of the 2-D Array class

The **Subtract** method, not shown here, performs a subtraction of each corresponding element of two arrays. Its code is nearly identical to that of the **Add** method.

The final method of the **2-D Array** abstract base class is the **Free** method, which empties the **arrayList**. Its code is depicted in Figure 13.39.

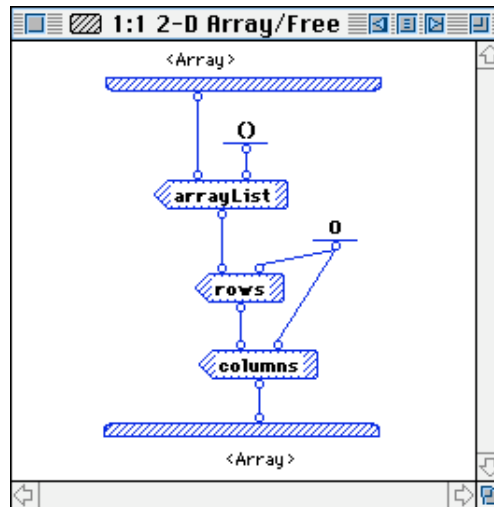


Figure 13.39: Free method of the 2-D Array class

It is actually the subclasses of the 2-D Array base class that we'll use. As with the Array, IntArray and RealArray classes constructed in Chapter 11, we'll provide the 2-D Array class with integer and real two-dimensional array subclasses. Since these two subclasses are very similar, we'll only show the class methods of the 2-D RealArray class here. The attributes of 2-D RealArray are all inherited from its parent class.

The instance method for this class first makes a list of real numbers to serve as a sublist, then constructs a list that contains several of these sublists. In other words, an inner list of real numbers is first constructed, then an outer list "wrapper" is built, made up of these sublists (see Figure 13.40).

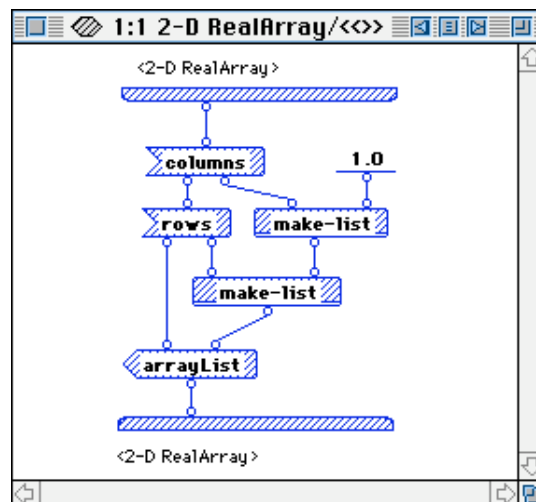


Figure 13.40: Instance method of the 2-D Array class

The **SetElement** method (Figure 13.41) overrides the corresponding method of its superclass so that type checking may be done on the new array element value. It calls the parent class' method to perform most of its actions.

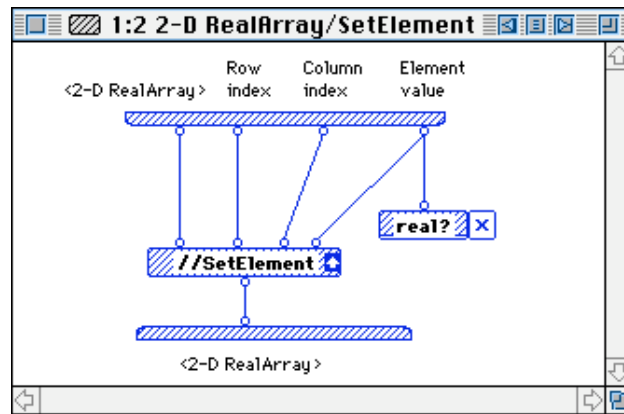


Figure 13.41: SetElement method of the 2-D Array class

The only other method that must be implemented in the subclasses of 2-D Array is the **Resize** class method (see Figure 13.42), which gives the array its dynamic properties, allowing the array to grow or shrink in size. This method, in turn, calls two additional methods -- **ResizeColumns** and **ResizeRows**.

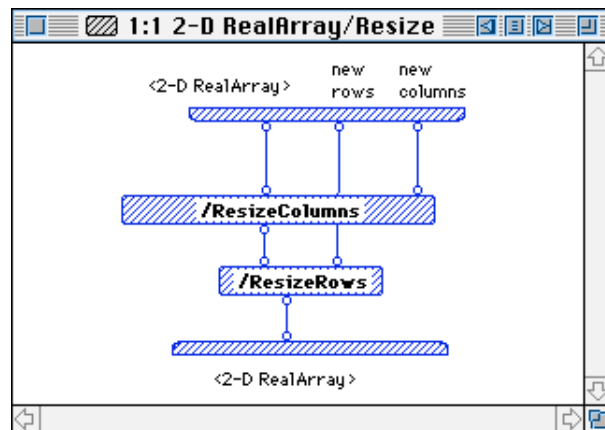


Figure 13.42: Resize method of the 2-D Array class

ResizeColumns class method is composed of a loop that is entered once for each row of the array, as shown in Figure 13.43. The loop calls the do the resize local method.

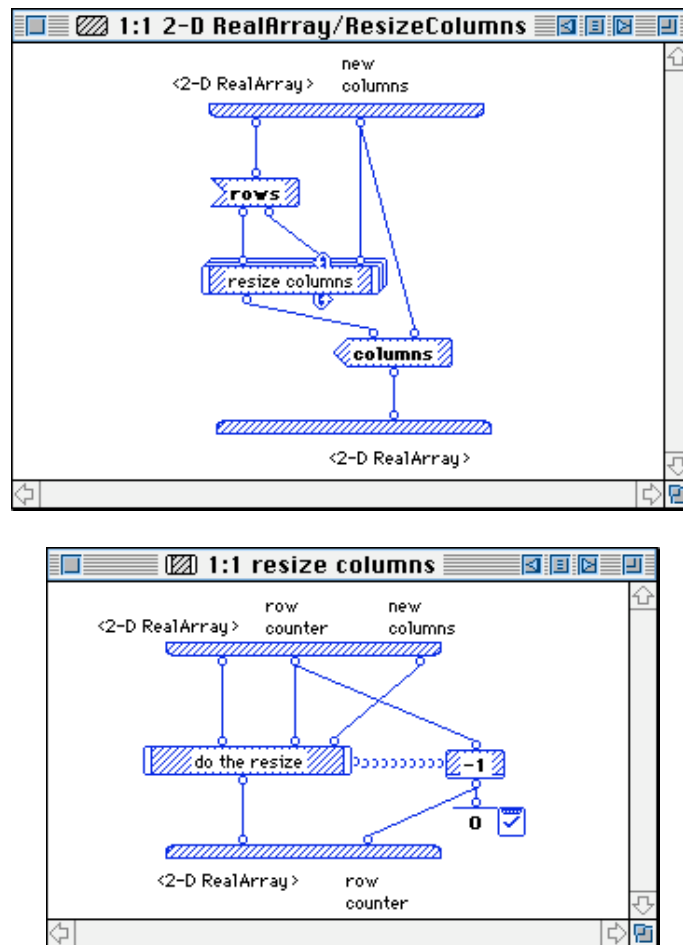
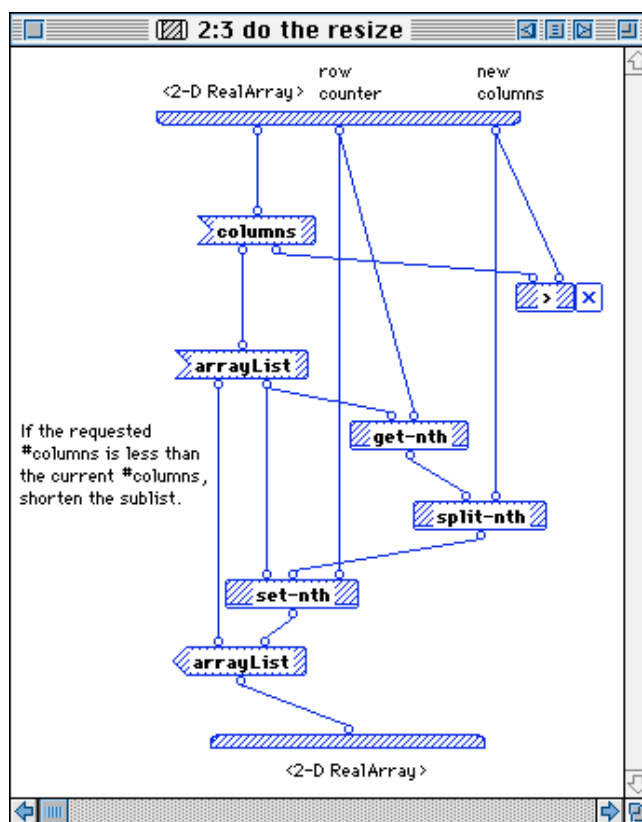
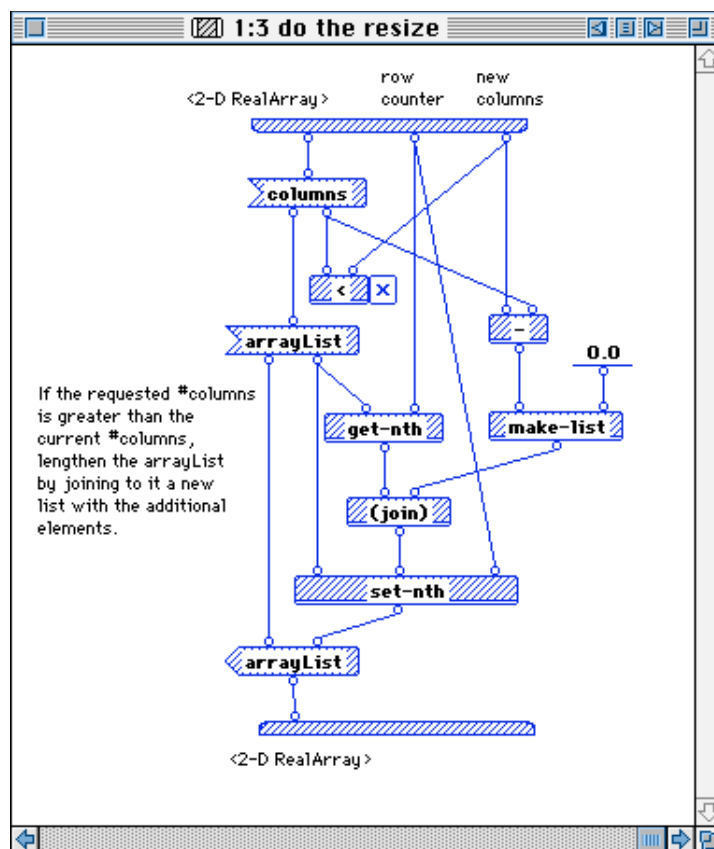


Figure 13.43: ResizeColumns method of the 2-D Array class

The do the resize local method (Figure 13.44) checks if the requested column size is larger, smaller or equal to that of the current column size. If it is larger, additional elements are joined onto the end of each sublist. If it is smaller, elements are chopped off of each sublist. If it is equal, nothing is done.



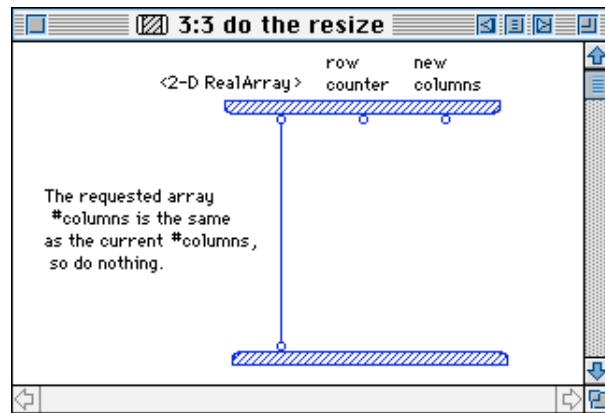
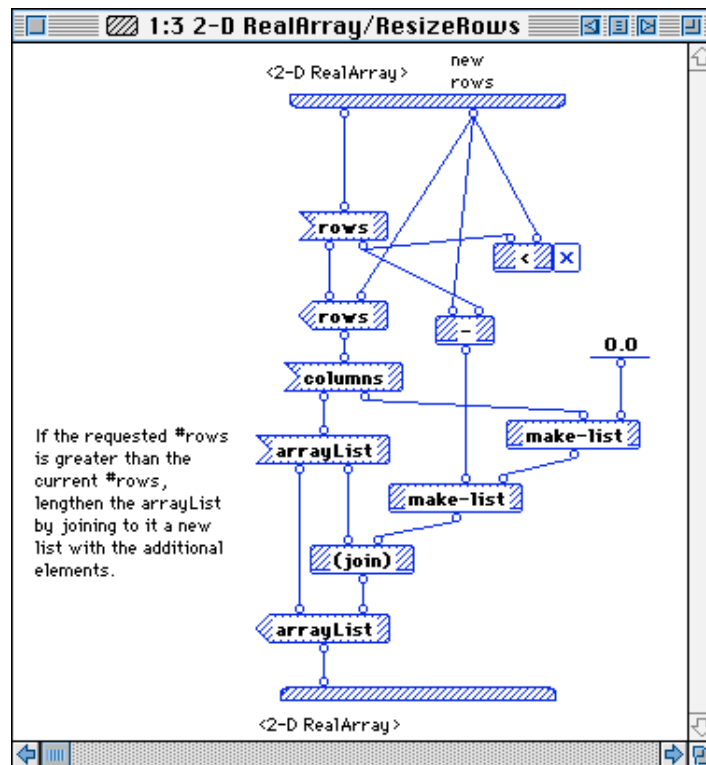


Figure 13.44: do the resize local method

ResizeRows (Figure 13.45), like **ResizeColumns**, checks if the requested row size is larger, smaller or equal to that of the current row size. In this method, if the requested number of rows is larger than the current **rows**, additional sublists are created and added to the **arrayList**. If it is smaller, sublists are deleted from **arrayList**. If it is equal, nothing is done.



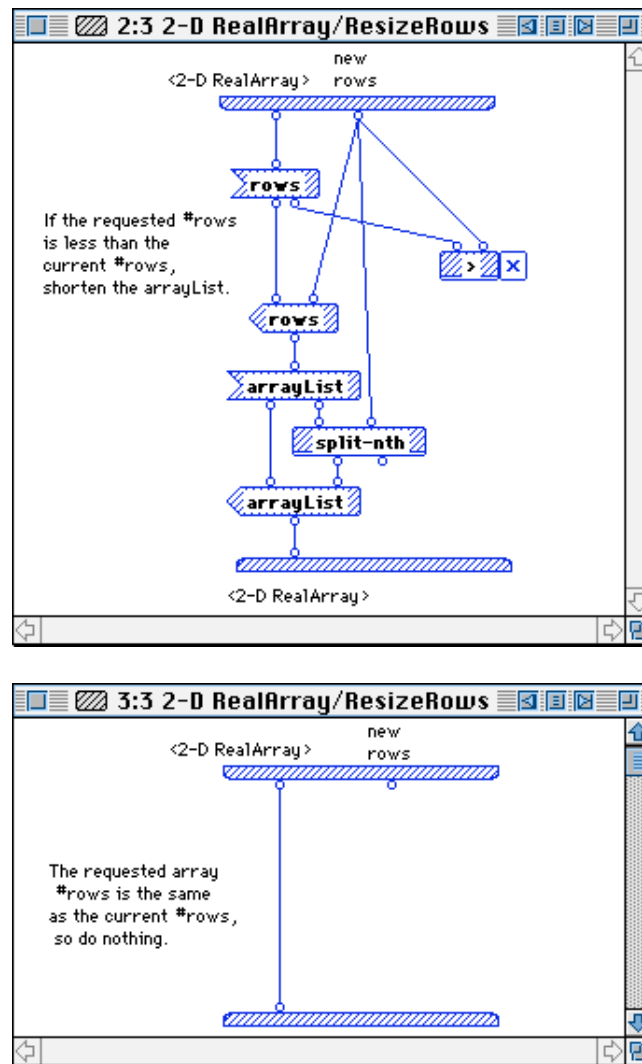


Figure 13.45: ResizeRows method of the 2-D Array class

We'll take advantage of the two-dimensional array classes to build the **Matrix** class. The major attribute of the class, **matrixList** (see Figure 13.46), will contain a two-dimensional array object. The **rows** and **columns** attribute determines the size of the matrix. Remember that in a square matrix, the number of rows equals the number of columns, so we only need one attribute to hold both values.

The **arrayType** attribute is extremely important. It will determine which type of array we'll use to hold the matrix contents -- a 2-D IntArray or a 2-D RealArray. In this way, we can mimic one recent addition to the C++ language -- *parameterized types* or *class templates*. These C++ classes are typically "container"-type classes that hold other data types. Templates allow these classes to hold any type of data. The exact data type is "filled in" at compile time.

In Prograph, we can perform a similar function by instantiating the **Matrix** container class with either an integer or real **matrixList**. Type checking will then be done on any new matrix element values via class methods.

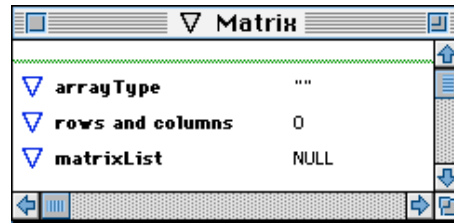
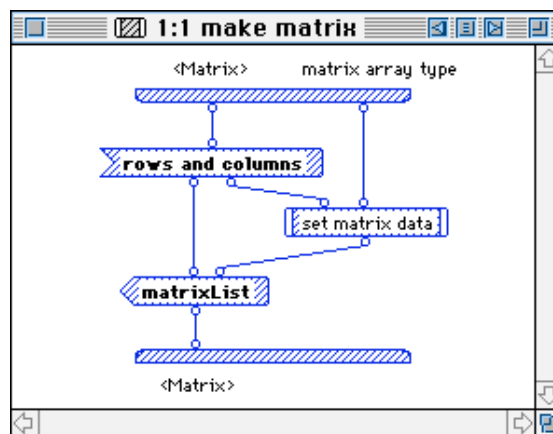
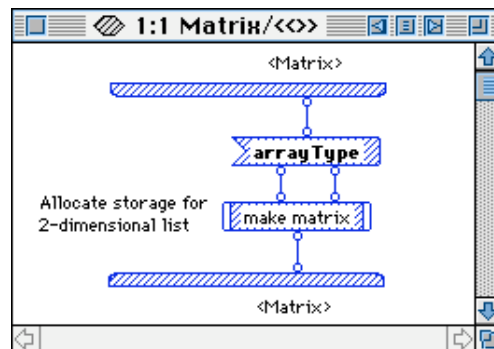


Figure 13.46: Attributes of the Matrix class

How does this template-like behavior originate? To answer this, we must look at the *instance method* for the **Matrix** class, shown in Figure 13.47. This method reads in a newly-created but as yet uninitialized instance of a **Matrix**. It creates a 2-D Array-derived object based upon the value of **arrayType**; that is, if an integer matrix is needed, an 2-D **IntArray** is created for the **matrixList**. Otherwise, a 2-D **RealArray** is created as the **matrixList**.



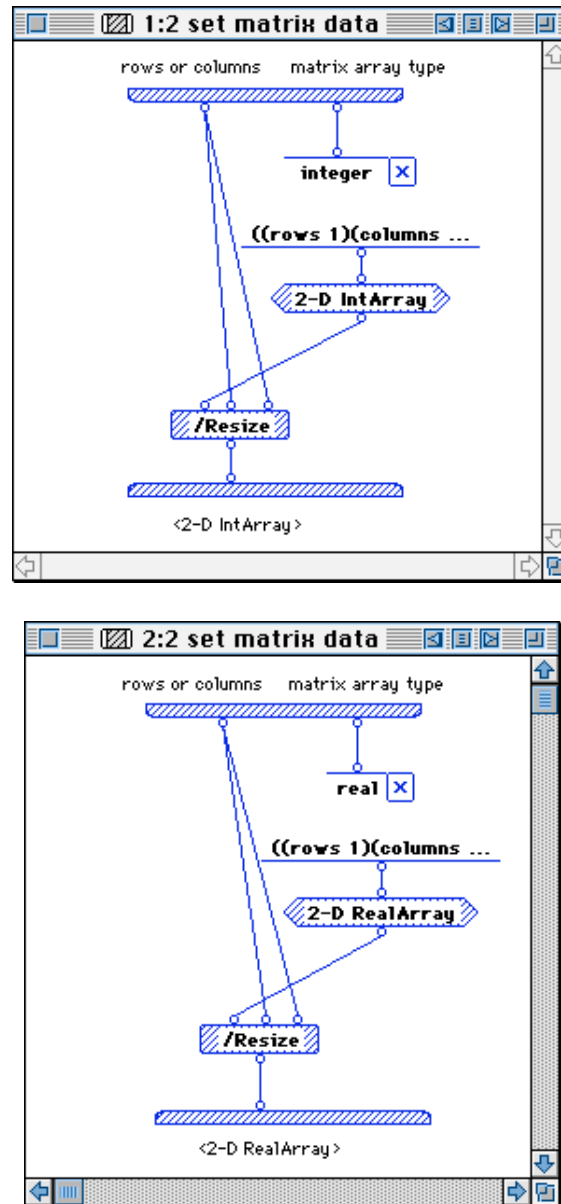


Figure 13.47: Instance method of the Matrix class

The corresponding **Free** method (Figure 13.48) acts much like a C++ destructor for this class. It deallocates the storage for the **matrixList** by replacing its contents with an empty list.

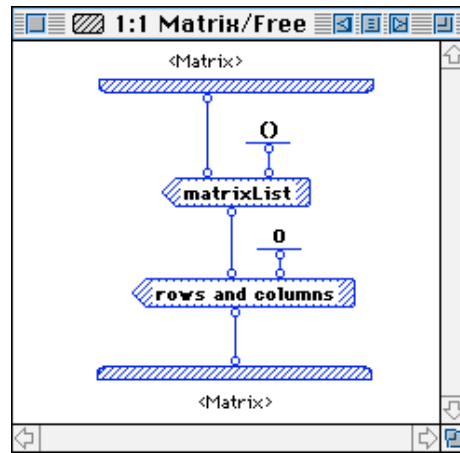
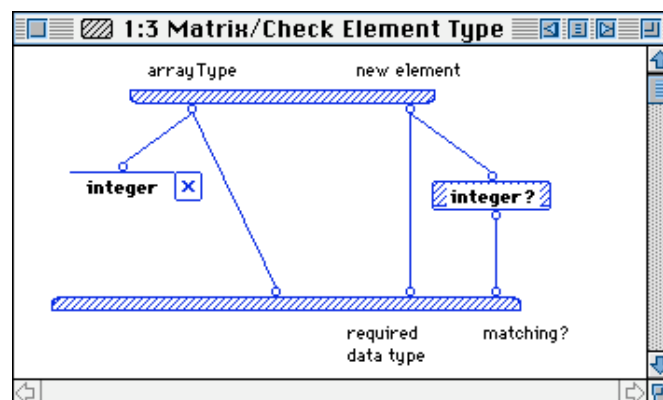


Figure 13.48: Attributes of the Square Matrix class

The `GetElement`, `SetElement` and `Copy` class methods are not shown here because they are similar to their counterparts in the 2-D Array classes. The `Copy` method simply calls the copy primitive to duplicate the matrix. The element access methods check that the requested element's row and column do not exceed the matrix bounds, then call the `GetElement` or `SetElement` method of the `matrixList`'s 2-D Array object to actually get or set the value of the matrix element. However, before setting the element, the `SetElement` method calls two additional class methods that provide typecasting for the new element value.

`Check Element Type` (see Figure 13.49) determines if the new element's data type matches that of the `matrixList`. It returns a Boolean value that signals whether or not typecasting must be done. If the data type is one for which no corresponding two-dimensional array type exists, an error message is displayed and the output Boolean set to `TRUE` so that typecasting is not attempted on the inappropriate data type.



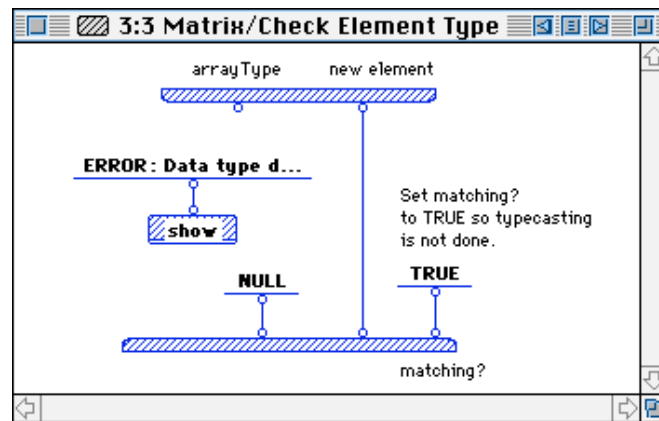
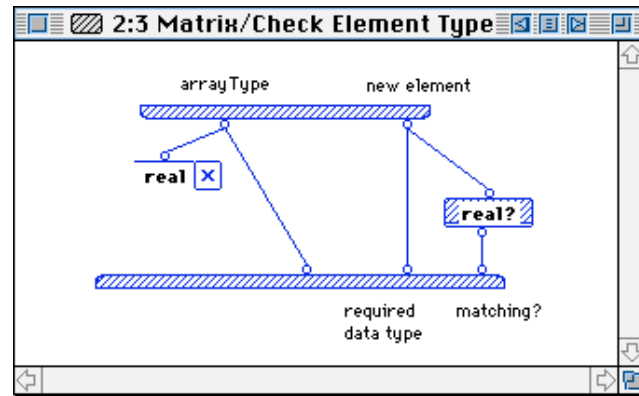
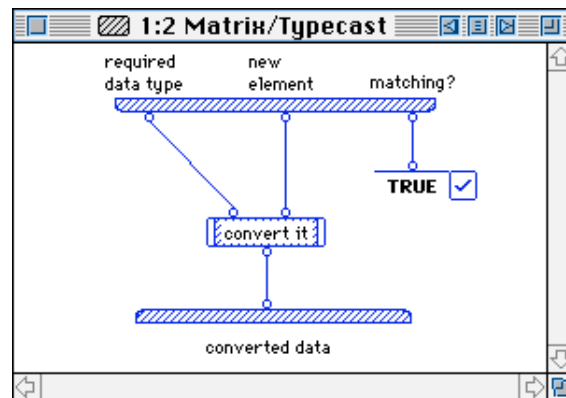


Figure 13.49: Check Element Type method of the Square Matrix class

The TypeCast method, shown in Figure 13.50, checks if the new element's data type needs to be typecast. If so, it is then converted to the proper data type; otherwise, it is passed through unchanged.



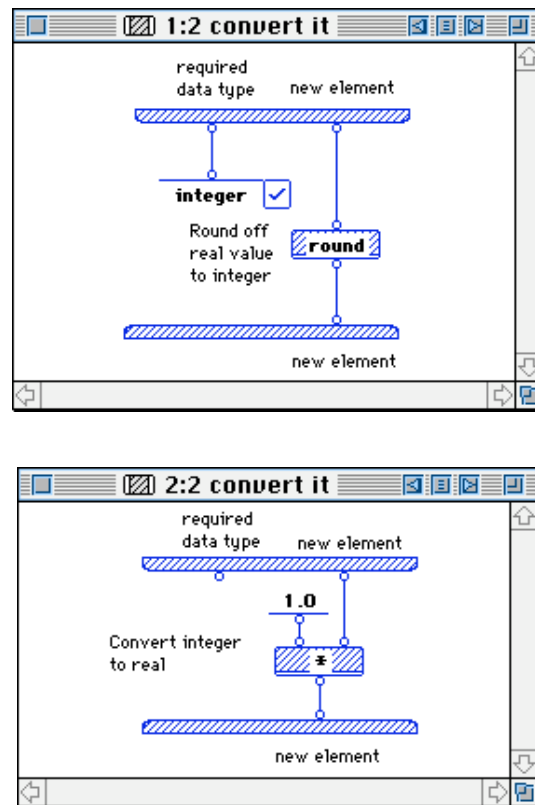


Figure 13.50: Check Element Type method of the Square Matrix class

The last general action of the **Matrix** class is to determine if two matrices are of the same dimensions. This is accomplished by the **EqualSize?** method (Figure 13.51), which simply compares the rows and columns attributes of two **Matrix** objects for equality.

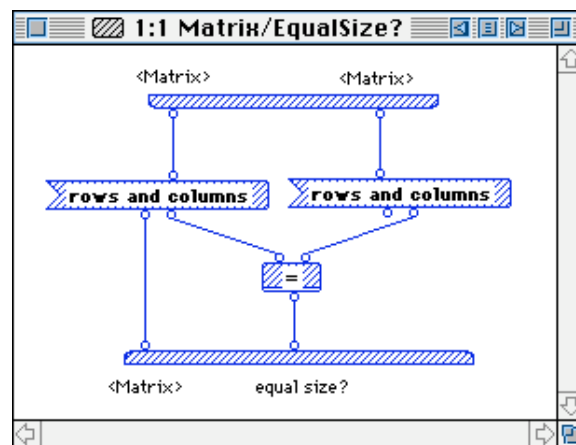


Figure 13.51: EqualSize? method of the Square Matrix class

Now let's get to the mathematical operations on matrices. The **Add** and **Subtract** methods are nearly identical, so we'll just show the **Add** method's code in Figure 13.52. This method gets the **matrixList** of each input matrix, then calls the **Add** method for the **matrixList**'s 2-D Array object. Wasn't that simple? This is why we built the 2-D Array class in the first place -- to greatly simplify the task of coding the **Matrix** class' methods. The matrix **Subtract** method performs the same actions, calling the 2-D Array/Subtract method instead. We'll discuss a **Multiply** method shortly.

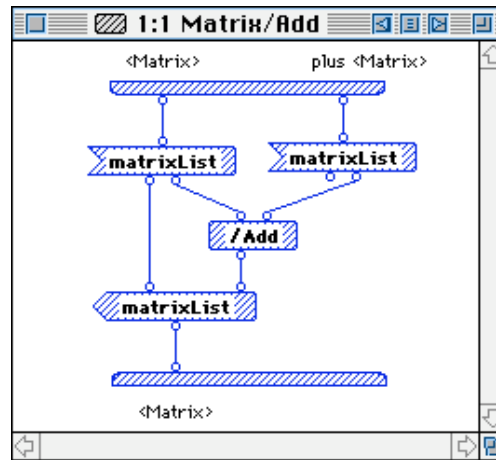
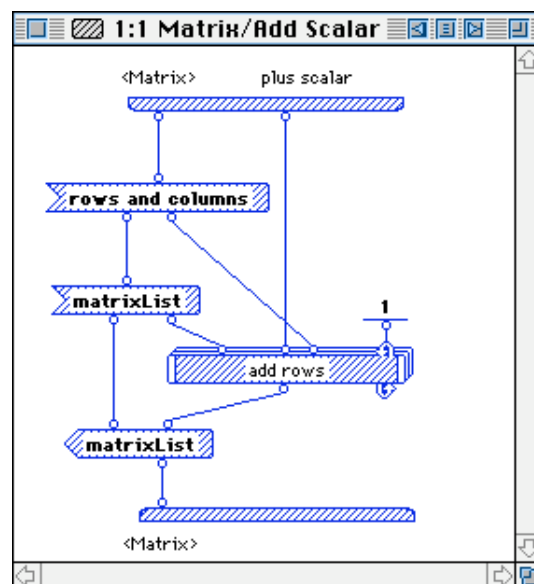


Figure 13.52: Add method of the Square Matrix class

Adding, subtracting or multiplying a matrix by a numerical scalar (non-matrix number) also share a common logical flow. The **Add Scalar** method, shown in Figure 13.53, enters a loop in which the scalar is added directly to each element of the matrix. The **Subtract Scalar** and **Multiply Scalar** methods are nearly identical.



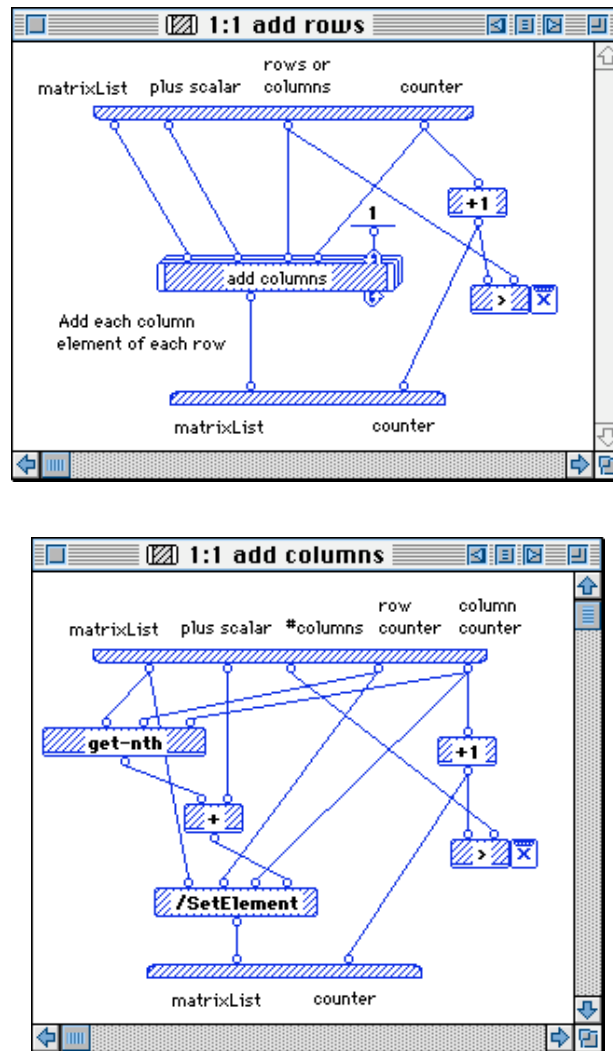


Figure 13.53: Add Scalar method of the Square Matrix class

Let's return to the **Multiply** method (see Figure 13.54). Here, we input the two matrices to be multiplied, then make a copy of one matrix to hold the product of the multiplication. A loop is then entered that calls the **Element Product** method once for each element of the matrices.

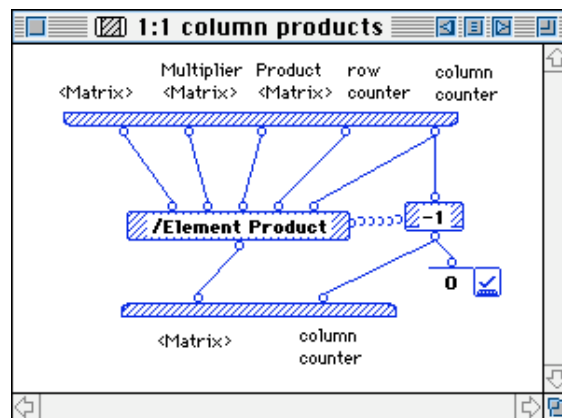
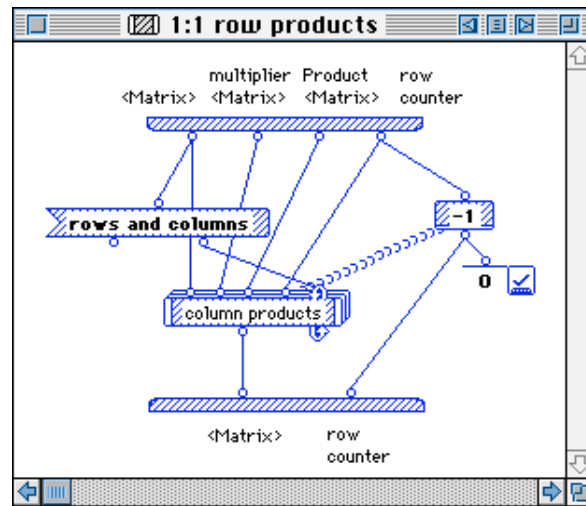
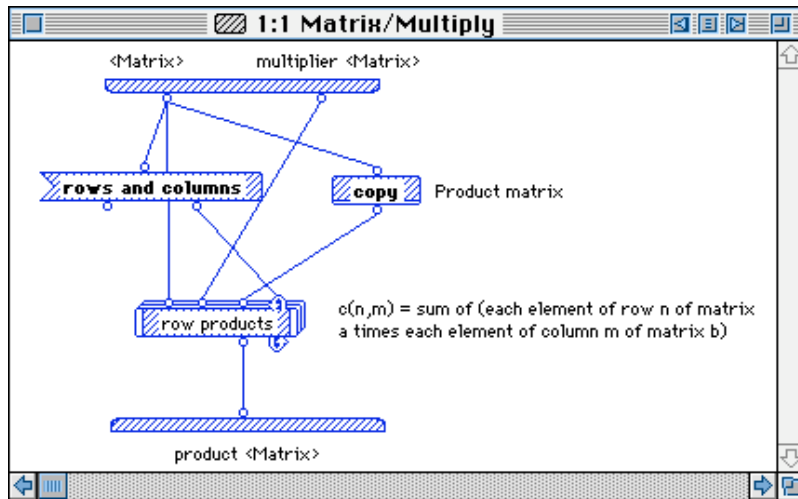


Figure 13.54: Multiply method of the Square Matrix class

The **Element Product** method (Figure 13.55) receives the row and column of the element to be multiplied, then sums together the products of each element in that row of the first matrix and each element in that column of the second matrix. This complicated sum is the product for that single matrix element.

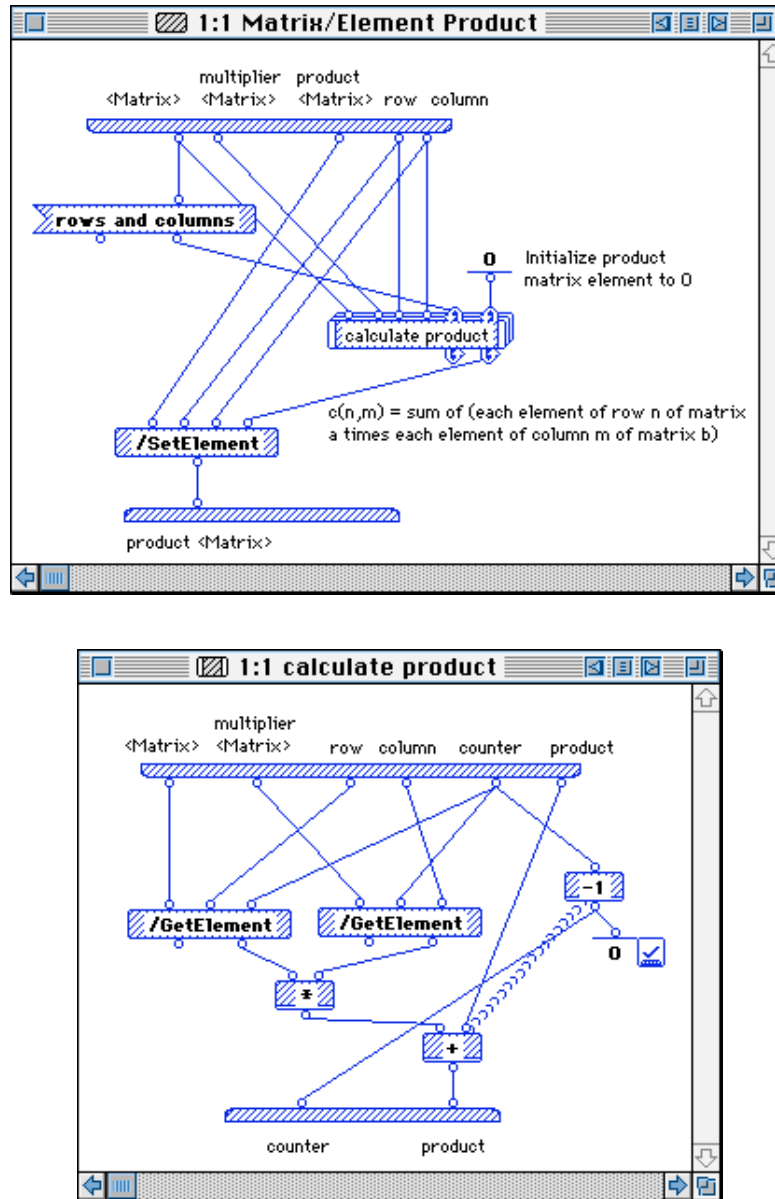


Figure 13.55: Element Product method of the Square Matrix class

This concludes the code for the **Matrix** class. The examples in this chapter should get you started in writing your own reusable classes. When applied carefully, OOP can supply you with a collection of classes that can be combined into new programs to considerably shorten your programming time.

Summary

In this chapter, we have constructed a number of useful classes that may be added to your Prograph programs. In the process, we have provided practical examples of many of the principles introduced in the previous chapters on object-oriented programming.

- Prograph classes may be used to build new data types such as the **Complex**, **Fraction** and **Matrix** classes that may then be called upon with little effort as if they were built-in data types.
- Many language features missing from Prograph but present in other programming languages like C++ may be mimicked. An example of this is the implementation of class template-like function in the **Matrix** class.

In Chapter 9, we suggested object-oriented programs could be thought of as systems of intercommunicating classes. While we could provide a simple example of how such a system might work, instead we'll show you an existing set of cooperating classes -- the Prograph CPX *Application Builder Classes*. These classes provide Prograph programs with a consistent user interface by sending method calls from one user interface object to another.