

11. Extending Classes -- Subclasses and Inheritance

Overview

The code we write to carry out a particular task is rarely needed only once in the same form. More commonly, we must perform similar but not identical tasks in other programs, or must modify a task for updated versions of a program. As an example, we may need one program to save *text* files, and another to save *graphics* files instead. In *procedural* programming, reworking the methods that handle text files to make new methods for *graphics* files could be tedious and error-prone. In many cases, one would be tempted to just throw out the code for handling text files and start programming again from scratch to make a program using graphics files. This can be a tremendous waste of time and effort.

OOP has built-in mechanisms for making changes in programs while minimizing the need to write new code. One such mechanism is *inheritance* -- the improvement or specialization of existing classes by creating new *subclasses* from them. In this chapter, we'll use inheritance to create systems of related classes that share code. We will also examine *composition* -- the reuse of existing classes by enclosing one object *within* an object of another class -- to add the features and actions of one class to those of another.

Subclassing (Inheritance)

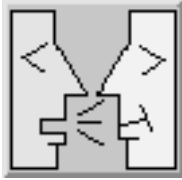
Subclassing is the addition of new functionality to a pre-existing class by deriving a new class from it. The new subclass automatically *reuses* any parts of the original *superclass* that are still useful. A subclass is defined solely by *additions* or *changes* from its superclass. This represents a substantial savings in coding effort for new tasks -- we can write only small changes in classes rather than starting all over again to suit each new situation.

In the file format example mentioned above, we could start with a general **File** class that would encompass most of the activities required for file reading and writing. We would then subclass the **File** class to produce a more specialized **TextFile** class for one program, and subclass it to make a **GraphicsFile** class for the other program. Each subclass would inherit general file-handling code from the **File** class. The programmer would only have to add the specific code needed to represent the *text* or *graphics* in a file -- the *changes* needed for the **TextFile** and **GraphicsFile** subclasses. All the rest of the code needed for each would already be present in the **File** superclass.

Let's progress to a concrete example. Prograph supplies you with a built-in *list* data type. Lists, you may recall, are *collections* of data elements. Data elements may be added to or removed one at a time from the list, and operations performed upon the list as a whole. You could mix different data types in a single list, such as integers and reals and Booleans, etc. Those who have programmed in other computer languages like C or C++

might be more familiar with the concept of an *array* -- collections of a *fixed* length that contain a *single* data type. All elements of the array are created *all at once*. Once created, you can access or set any element within it in any order. *Dynamic* arrays are a special type of array whose length can be *resized* should the need arise, say, if the array holds 50 items but now you need to hold 55 items.

In this chapter, we will create a generic dynamic array class called **Array**, then subclass it to produce classes for arrays of integers or real numbers. This is where the power of object-oriented programming shines -- the ability to create new data types that are not supplied by the programming language itself. In this case, we'll use OOP to mimic a data type that is present in other computer languages but not Prograph. Our generic dynamic array class will provide the ability to create an array, destroy it, initialize the elements within it to known values, resize the array to a larger or smaller length, and get or set the value of any individual element. The **RealArray** subclass will store real numbers as the elements in the array, while the **IntArray** subclass will do the same for integers. New subclasses such as **OffscreenArray** (an array of off-screen graphics representations) could also be built just as easily from the **Array** parent class.



By The Way...

Why do we need separate subclasses for integer arrays and floating-point arrays? In languages such as C++, separate data types are needed to store integer and floating-point arrays because each requires a different amount of memory storage and a different “spacing” between one element of the array to the next in memory. Different array types allow each type of data stored in the array to be stored and accessed as efficiently as possible. But in Prograph, both a *list* could be used as an attribute to store the array elements. Lists *don’t care* what type of data they contain, so we should be able to store either integers or floating-point numbers in the same list. Lists would handle both element storage and access for us *regardless* of the type of data the list contained -- the data will be stored and accessed the same way for both types of data. Wouldn’t this mean that a single **Array** class could serve to hold either integers or real numbers? Actually, the advantages of lists are also a liability -- the reason why we *do* need to subclass the **Array** class for each data type. How do you ensure that the list will hold a *single* particular data type if you want it to, that is, only integers or only reals? Remember that Prograph is an *untyped* language. The list won’t check whether or not the type of data you are trying to stuff into it is correct or not. It is up to *you* to do data type-checking yourself! If the list attribute that stores your array elements sends its data to fast *integer-only* math primitives and you didn’t ensure that the array contained *only* integers, your program will result in an error. We must perform type-checking on the data stored in the array’s elements by writing specific subclasses of an **Array** class that will each contain class methods that will ensure the storage of only the appropriate data type for that class. That is, an **IntArray** subclass will enforce the storage of only integers while a **RealArray** subclass will enforce the storage of only real numbers.

Abstract Superclasses

Start by creating a new section called Arrays. We will store our three array classes (**Array**, **RealArray** and **IntArray**) in this section so that we can reuse them later. Get used to storing classes and their descendants in sections. You can add these sections to later programs to reuse the classes. A similar process is used for easily handling user interfaces for your programs by loading sections of pre-written classes called the Application Builder Classes (see Chapters 14-16) into your own programs.

Create a new class in the Classes of “Arrays” window and call it **Array**. This will be an *abstract superclass* -- a special class that is never used to create objects itself, but *exists solely to be subclassed*. You never use objects of abstract superclasses themselves, so we must prevent the superclass itself from creating its own array elements. The data

type of the array elements is set in the *subclasses* of **Array** -- only **RealArray** and **IntArray** can create array elements, specifically of type real or integer. Since each type of array can only contain a single data type, each array type needs to check what kind of data it is receiving when setting an array element, or when copying another array into itself. The code that checks data type therefore is also specific to each subclass. This is why **Array** must be an *abstract superclass* -- too many of its implementation details can't be specified until we make subclasses from it.

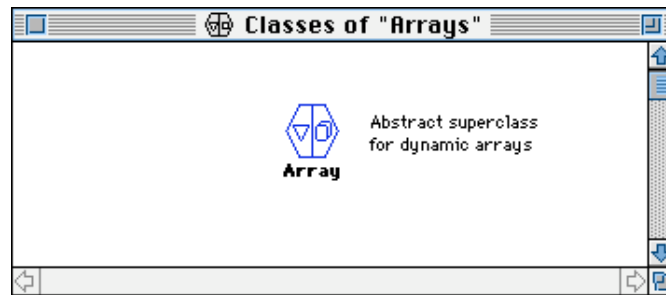


Figure 11.1: The Array abstract superclass

Open the Attributes window of the **Array** class and create two attributes. The first will be an integer called **length** that will keep track of how many elements are in the array. The second will be a *list* named **arrayList** that will hold the array elements. We are designing the **Array** class to create a new data type that *extends or modifies the behavior of a pre-existing data type* (a *list*) to fit a new set of applications.

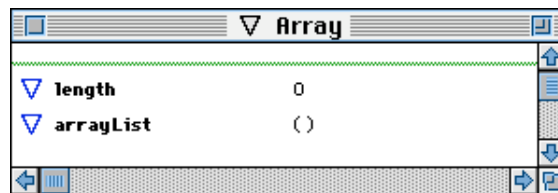


Figure 11.2: Attributes of the Array superclass

What happens when we no longer need to use the **Array**? We remove all of the array elements from the **arrayList**. The **Free** method (Figure 11.3) accomplishes this task by setting the **arrayList** to an empty list, then resetting the value of **length** to 0. This is accomplished the same way for any type of array, so the **Free** method may remain in the **Array** parent class.

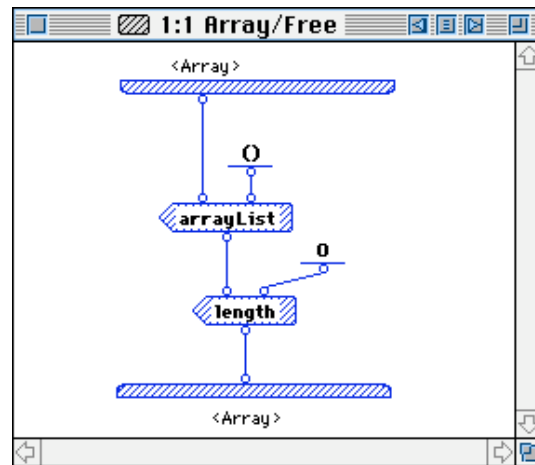


Figure 11.3: The Free method

Two important class methods for the **Array** class are the **GetElement** and **SetElement** methods, which access and set one element of the array, respectively. You may ask “The array is stored in the **arrayList** attribute -- couldn't I simply use a *Get* or *Set* method to read or write an element of the array in my programs?” Well, yes, you could. OOP purists would argue that using a *Get* or *Set* method directly would allow your programs that use the **Array** class unwanted access to the inner workings of the class. Remember that one advantage of OOP is *encapsulation* -- insulating the user from the details of how the class is constructed and coded. If the user of the class directly accessed the attributes anywhere in their program, mistakes could be made. Ideally, the user should only access the class with methods we’ve defined as the *class interface*. So long as the user calls only those methods in the class interface, they won't accidentally corrupt the **arrayList** contents. *Get* and *Set* methods are really meant to be used only *within other class methods*, not by the user of the class. To preserve the integrity and safety of the class interface, we’ll provide class methods to give the user a way to read or write to array contents without directly using *Get* or *Set*. These methods will then in turn call the **arrayList** *Get* and *Set* methods *within* their own code, hidden from the user.

There’s a second reason why we include our own **GetElement** and **SetElement** method, but this reason is more practical. We could easily use a *Get* method to access array data and then just reuse this code in both array subclasses, since a *Get* method would return *whatever* type of data we’ve stored in the array. However, we are responsible for providing our own individual methods to *set* the array elements for each subclass, or else the arrays might not hold one consistent type of data. If we just wrote a user-defined **SetElement** method, it would use different data access techniques than would a built-in *Get* method. For consistency of class access, we’ll write both **GetElement** and **SetElement** methods so that each uses the same technique to access the array’s list data. The user can therefore read and write array data using consistent syntax.

The degree of encapsulation afforded by writing our own **GetElement** and **SetElement** methods also allows us to do *automatic* tasks every time an array element is read or written. Remember that the use of arrays poses the risk of trying to access array members past the end of the array. In the **GetElement** method, whose first case is shown in Figure 11.4, we can first check if the user is trying to access an empty array or an element beyond the end of the array. If so, an error message is displayed by a second case.

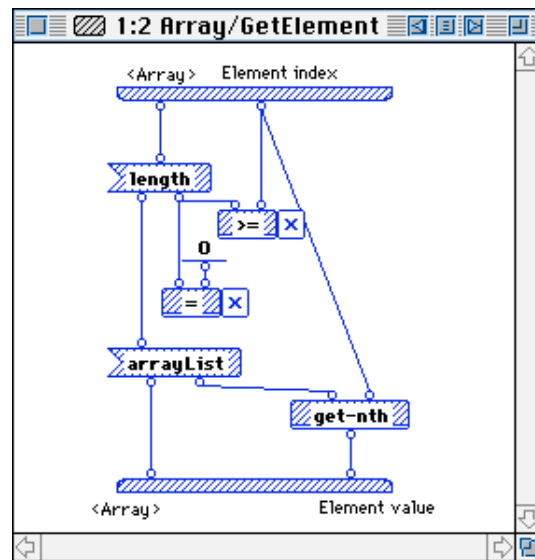


Figure 11.4: First case of the GetElement method

Similarly, **SetElement** (Figure 11.5) also checks if the user is trying to write an element to an empty array or beyond the end of the array, and displays an error message. This method replaces the contents of an element of the **arrayList** with a new value by calling the **set-nth!** primitive. Notice that this method does not care what type of data the new value of the element should be. This data type checking will be handled by the subclass, as we'll see shortly.

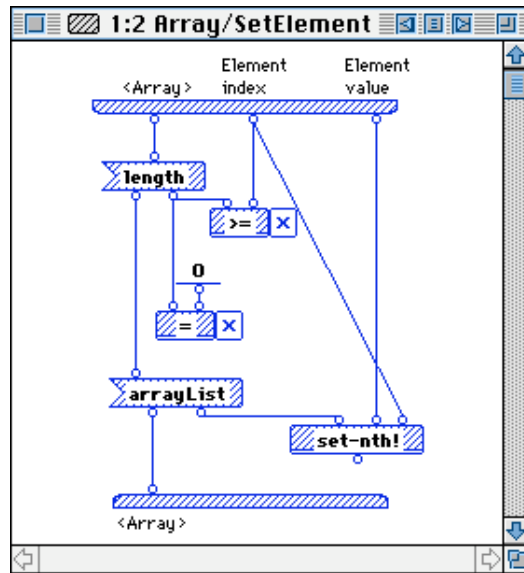


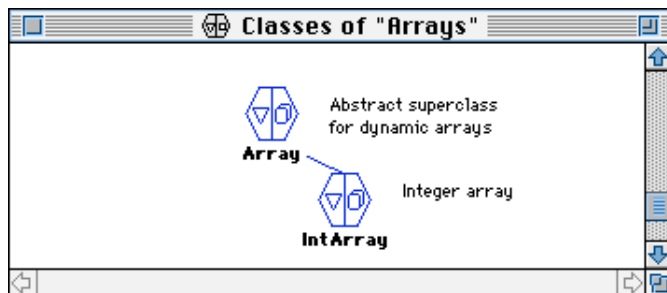
Figure 11.5: First case of the SetElement method

The Array class will contain two more methods. We'll defer discussion of these two methods for now.

IntArray -- The First Subclass of the Array Class

Now comes the fun part. We will *subclass* the general-purpose Array class to form two new classes for storing arrays of integers and real numbers. The same principles can be used by you to make other new array types for Booleans, Macintosh data types such as Pictures, etc.

Return to the Classes window. Highlight the Array class icon. While pressing the *option* key, create a new icon below the Array class icon. Name this new class icon IntArray. The link between the two class icons, analogous to a datalink, shows that the IntArray class is a *subclass* of the Array class, as shown in Figure 11.6. Once again, visual programming makes program components easier to understand. The link between the classes in the Classes window makes their superclass-subclass relationship obvious. In the equivalent C++ code shown, this relationship is less obvious since the declaration of both the superclass and the subclass looks so similar.



```
class Array {
    // Class contents
};

class IntArray :
    public Array {
    // Subclass contents
};
```

Figure 11.6: Subclassing the Array class to make a new IntArray class, shown with its C++ language equivalent

Open the Attributes window of the IntArray class by double-clicking the left side of the class icon (see Figure 11.7). Notice that the IntArray class has come equipped with the same attributes as the Array class. These attributes have been *inherited* automatically from the Array class when you subclassed IntArray from Array.

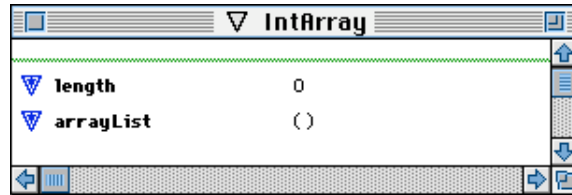
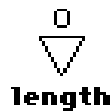


Figure 11.7: Inherited attributes in the IntArray subclass

There is one difference between IntArray's attributes and Array's attributes. The symbols for the attributes are different. The arrow within IntArray's attribute symbols reminds you that these attributes have been inherited from the *superclass* of IntArray (Figure 11.8). In Prograph, you can examine all of the attributes in a subclass, yet still keep track of which attributes are inherited and which are specific to the subclass. This differs markedly from the notation used in C++, where inherited data members are *not listed at all* in a subclass -- the programmer must remember the data members of the parent class.

Original Attribute
in class Array



Inherited Attribute
in class IntArray



Figure 11.8: Inherited attributes

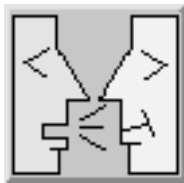
Instance or Initialization Methods

The first method we will write for the IntArray class is a special optional method used solely when creating objects from classes. When we create a new object, the object's attributes' initial values are set to the default values we give them in the class definition. But what if we *don't know the value of an attribute* until run-time? We certainly can't just use the default value -- it may not always be the value we'll need. Luckily, we have an easy way to set attribute values when we create an object. We do this by writing our own *initialization or instance methods*. These methods are called

automatically right after you create (instantiate) an object. Within the initialization method, you can include code to set any attributes you want and override their default values, or allocate memory. In this respect, instance methods are the counterpart of C++ constructors, which automatically initialize C++ classes. However, Prograph does not provide the equivalent of C++ *destructors*, which are called when an object is destroyed. Any memory allocated by a Prograph initialization method must be freed *explicitly* when the object is no longer needed. It is the programmer's responsibility to remember to do this by writing a method to do so which will be called when the instance will no longer be used.

We will create an initialization method for the `IntArray` class that will set the number of array elements contained in each `IntArray` object at runtime, then allocate enough memory for the `arrayList` attribute to hold these array elements. Create a new class method in the Classes window. While the new class method icon is still highlighted, select the Instance menu item. The method icon changes to an *Instance icon*, showing that it's an *Instance* or *initialization* method. The name of an initialization method is always set to “<>”.

Remember what makes the `IntArray` class different from the generic `Array` -- `IntArray` stores only *integers*. When we need to create elements to store in the `arrayList`, we must create *integer* data. The `Array` class, an abstract superclass, was prevented from creating any elements of its own (since their type would be unspecified) by purposefully omitting an initialization method for that class that would fill the `arrayList` with elements. For the `IntArray` class, we'll want to create integer elements, so we'll write an initialization method for it that explicitly creates a list full of integers.



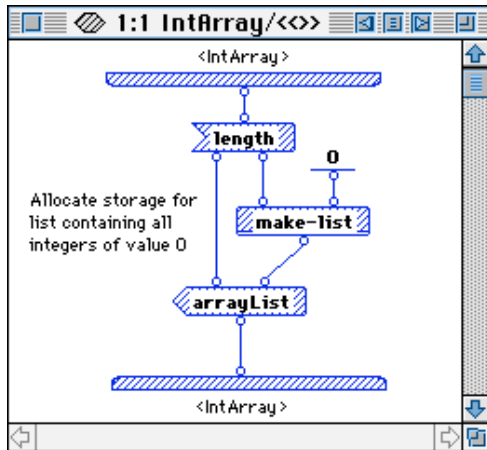
By The Way...

Although we could not prevent the user from *instantiating* an `Array` (generating an `Array` object), we did the next best thing -- we prevented the user from doing anything practical with the newly-created `Array` object by not permitting an empty `arrayList` to be filled or accessed.

Open the new initialization method and complete its code diagram as shown below in Figure 11.9. This method simply sets the **length** of the array and calls the `make-list` primitive to create an entire list all at once. The second input (the integer 0) to the `make-list` primitive tells the primitive to initialize each element of the `IntArray`'s `arrayList` to the *integer* zero. This initialization method is doing more than just setting the value of an attribute at run-time. It is also *automatically allocating memory for the array's elements* and *setting their data type* when the `IntArray` object is created. This is what makes initialization methods so powerful.

Another powerful feature of Prograph instance generators is that they accept one very important input -- a list containing values for the attributes of the object to be created, which provides another way in which they can be initialized. Why is this so important? This feature allows us to specify and set the **length** attribute that determines

the number of elements of the `arrayList` *at the time that we create an object* of type `IntArray`. If we input an attribute initialization list to the `IntArray` instance generator of `IntArray`, the value of the newly-formed `IntArray`'s `length` attribute can then be used by the initialization method we've written. For example, if the attribute initialization list contained an element setting the `length` attribute to 10, the newly-created `IntArray` object would be initialized by our initialization method to have a `length` of 10 and an `arrayList` containing 10 elements. We'll see how this works when we put the `IntArray` class to use.



```
class IntArray {
public:
    IntArray( short length
);
private:
    int *fArrayList;
    short fLength;
};

IntArray::IntArray(
    short length )
{
    fLength = length;
    fArrayList = MakeList(
        fLength, 0 );
}
```

Figure 11.9: The Instance or initialization method for the `IntArray` class with an equivalent C++ constructor

Changing the size of an existing `IntArray` now becomes an easy process, since it's very similar to the steps we've taken in the initialization method. The `Resize` method (Figure 11.10) checks the current value of the `length` attribute and compares it to the requested new length. If the new length is larger than the current length, the `make-list` primitive is used to add new elements to the `arrayList`. It creates enough new elements to cover the difference between the new length and the old length.

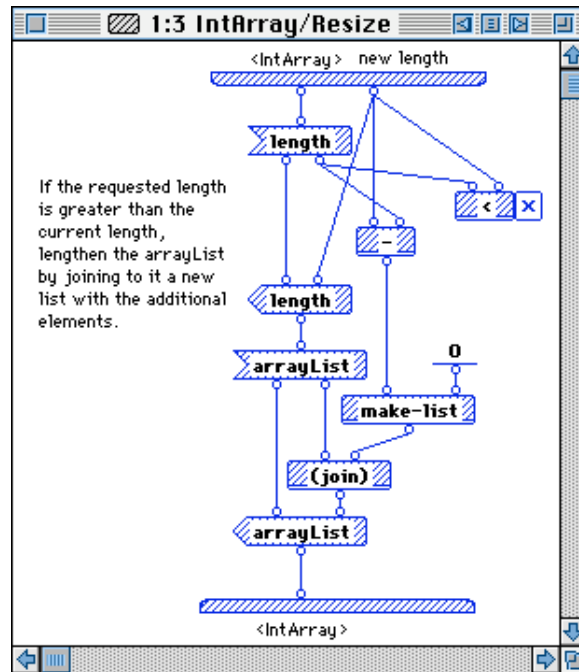


Figure 11.10: First case of the Resize method

If the new length is shorter than the current length, a second case is entered in which the `arrayList` is partitioned with the `split-nth` primitive to “slice” the array at its desired length and remove the proper number of elements from the `arrayList`. This second case is shown in Figure 11.11.

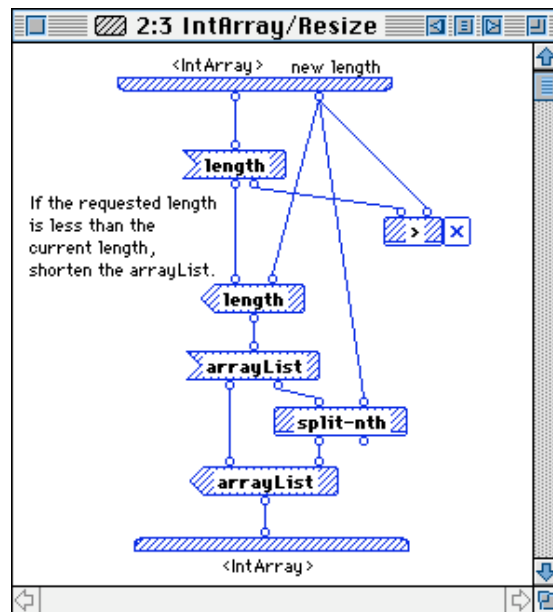


Figure 11.11: Second case of the Resize method

Finally, just in case the user accidentally requests a new length that's just the same as the old length, we enter a third case that appropriately does nothing.

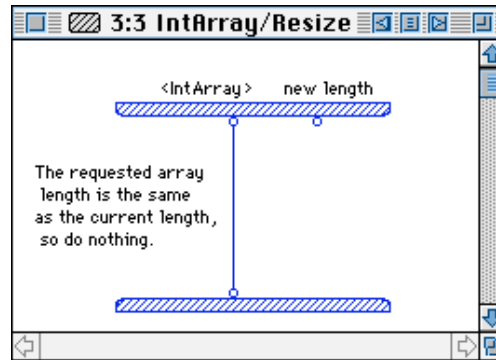


Figure 11.12: Third case of the Resize method

Overriding and Accessing Superclass Methods

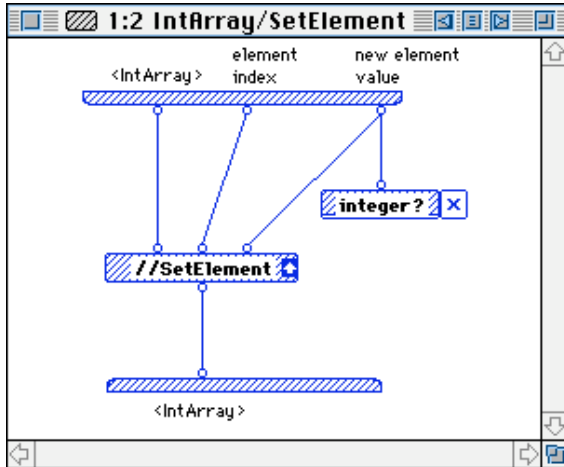
When we subclass, do we have to rewrite *every method* of the superclass to provide behaviors for the subclass? Of course not! This is one of the main reasons for using object-oriented programming. We only have to rewrite those methods that will specifically *change the behavior of the subclass* relative to its parent class. The remaining methods -- in which the subclass *does the same thing* as its superclass -- are simply *inherited* and *reused* as is without rewriting. If we *did* need to rewrite every method, there would be no reason to subclass, since we wouldn't save any time or effort. Remember that another benefit of method inheritance is that if a method worked correctly in the superclass, it will also work correctly in the subclass that inherits it. This means that we'll spend less time fixing potential errors in the subclass' methods.

Prograph will *automatically* inherit all of the superclass' methods for us when we create the subclass, just as it did for the attributes. All we create in the class methods window of the subclass are either *new methods* or *methods to replace methods of the superclass*. We give the replacement methods *the same names as their corresponding methods in the superclass*. This process is called *overriding* or *overshadowing*.

All we need to override are methods that must be changed if we are to give the subclass different functionality from its parent class. Let's override the **SetElement** method for our **IntArray** class. Remember, we only have to write *some* of the methods from **Array** -- the ones that will make an **IntArray** different from an **Array**.

When we want to set new values for the integers stored in **IntArray**'s **arrayList**, we must first check that the value we're trying to put into the **arrayList** *really is an integer* before storing it in the array. We can do this by *overriding* the **SetElement** class method, adding code to check for the correct data type.

Create a new **SetElement** method in the **IntArray**'s class method window. Open its case window and complete it as shown in Figure 11.13. Note that the C++ equivalent code shown in Figure 11.13 isn't quite correct -- we would not want the parent class' *SetElement()* function to accept an integer argument.



```
class Array (
    // Class contents
    void SetElement( int );
};

class IntArray :
public Array (
    // Class contents
    void SetElement( int );
};

void IntArray::SetElement(
    int newValue )
{
    if (sizeof(newValue) ==
        sizeof(int)) {
        // Call superclass
        method
        ((Array*)this)->
        SetElement( newValue
    );
    }
}
```

Figure 11.13: The overridden SetElement method and its C++ equivalent

The *integer?* primitive checks the type of the incoming data to see if it's really an integer. If not, we enter a *second* case (not shown) that displays an error message. If the data *is* an integer, we need to store its value in the appropriate "slot" of our integer array. But how do we store the data? We could write code in this method to store the data. But wouldn't that be wasteful? We already have code in the **Array** class' **SetElement** method that stores data in the **arrayList**. Why not just *reuse* that code?

Prograph allows you to reuse code from a subclass' parent class by using a special *superclass method call*. It is set up by highlighting the desired method that you want to call from the superclass in the current code window and selecting the Super menu item. This converts an ordinary class method call to a *superclass method* call, as shown for the call to the parent class' **SetElement** in Figure 11.13. The reuse of the parent class' code saves us time. We'll reuse this method once again when we write a second subclass -- **RealArray** -- which will save us even more programming time.

Reusing parent class code also saves us time when debugging by helping us find potential bugs easier. Any potential problems encountered in setting array elements would have to be localized solely within the **Array** class' **SetElement** method, since this method is called by any subclass of **Array** to perform this task.

One Form of Polymorphism in Subclasses

Subclassing isn't used only to change the *existing* behavior of a superclass. It can also be used to *add new actions* to a class. We can *modify* the function of class methods belonging in a parent class by using code in subclasses. Let's do it now. We'll create two new class methods in the **Array** superclass -- **Add** and **Copy**. **Add** will add two arrays together, element by element. **Copy** will copy all of the elements of one array into the **arrayList** of a second array.

We'll start with the **Add** class method (see Figure 11.14). The **Add** class method first checks the length of the **Array** we wish to add to our original **Array**. It then calls the **Resize** method to change the length of the original **Array** if the second **Array** is a different size. Once we're sure that the **arrayList** of the two **Arrays** are of the same length, we add their elements together with the **AddElements** local method.

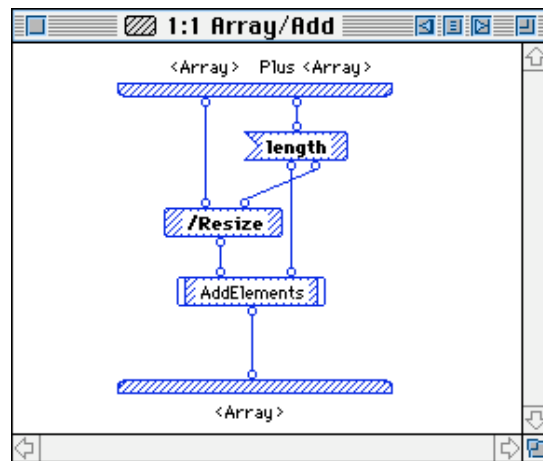


Figure 11.14: Add method of the Array class

The **AddElements** local method gets the size of the original **Array** (now equal to that of the second **Array**), then enters a loop that repeatedly calls the **AddOneElement** local method until each element for the entire length of both **Arrays** has been added together.

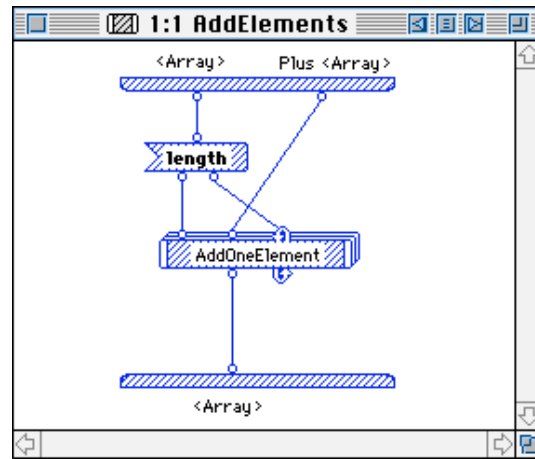


Figure 11.15: AddElements local method

AddOneElement simply gets the value of the array element at the same index from each **Array**'s **arrayList**, adds them together, then stuffs the result back into the same array element of the original **Array**. It starts with the *last* element of each (at index **length**), then counts *backwards* to step through the entire **arrayList** of each.

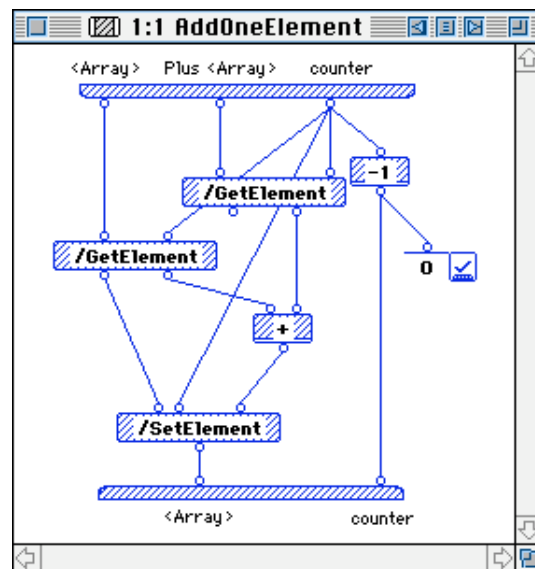


Figure 11.16: AddOneElement local method

That was pretty easy, wasn't it? The **Copy** class method works almost exactly the same as does the **Add** class method. Therefore, instead of explaining it in detail, we'll just show its code windows in Figure 11.17.

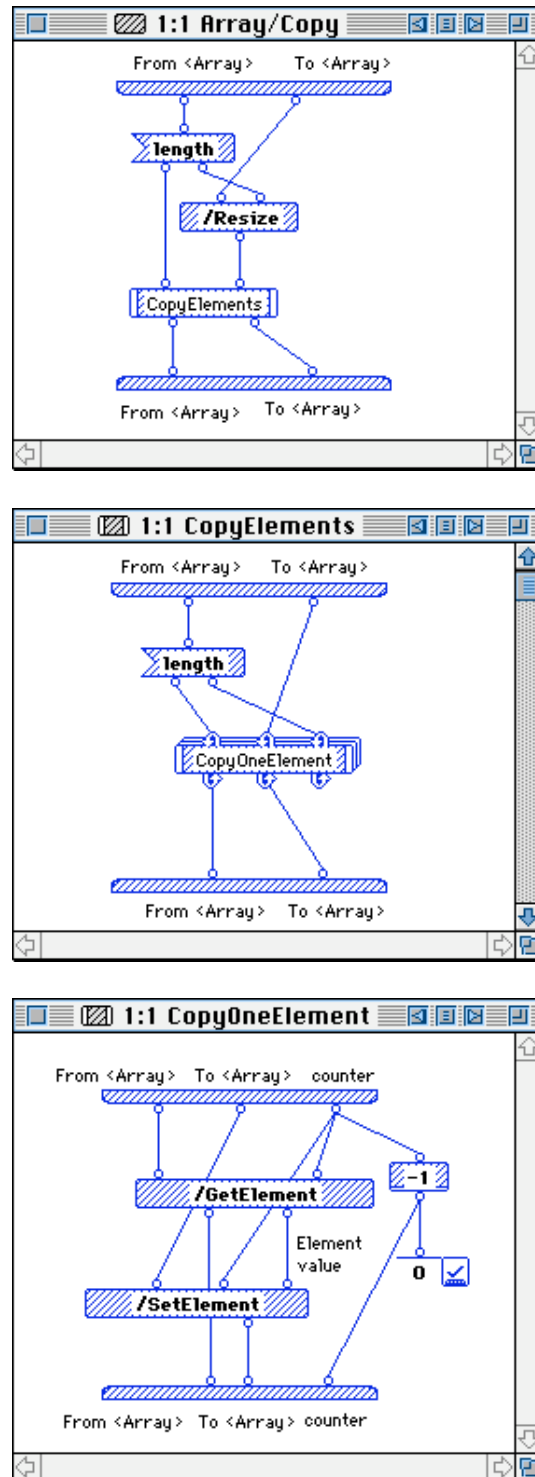


Figure 11.17: The Copy method of the Array class

What happens when an IntArray object's Add or Copy class method is called? Let's look at the Copy method in Figure 11.17. This method first reads the length attribute of the IntArray object. It then calls the CopyElements local method, which next

enters the `CopyOneElement` local method. What happens inside this local method? The `GetElement` class method is called. But the calling `IntArray` object doesn't have a `GetElement` method! What code is executed? Simple -- the `GetElement` method that `IntArray` automatically inherited from `Array`, since this is the only version of `GetElement` that exists -- the only possible version of `GetElement` that it could execute. The execution of the `Add` or `Copy` class method by a subclass of `Array` is therefore fairly straightforward.

On the other hand, let's look at what happens when the `SetElement` method is called within the `Copy` class method? Unlike `GetElement`, there is *more than one* version of the `SetElement` method -- one in the `Array` class and one in its `IntArray` subclass. Which version of `SetElement` will actually be executed? If an `Array` object's `Copy` method were to be called (that is, an `Array` object was the second input to the `Copy` method), the `Array` parent class' version of `SetElement` will get called. If an `IntArray`'s `Copy` method is called (an `IntArray` object was the second input to the `Copy` method), the *IntArray subclass' version* of `SetElement` will get called by `Copy`. This is an example of polymorphism at work. All the object (whether it's `Array` or `IntArray`) knows is that it needs to copy array elements with the `Copy` method. The appropriate code will be executed by `Copy` for whatever type of array that receives the method call.

Creating a Second Array Subclass -- RealArray

We now have a functional `IntArray` subclass that inherits much of its behavior from its superclass, `Array`. What would happen if we now had to write a new program that needed an array of real numbers instead of integers? In procedural programming, we'd have to write a lot of new code. But with OOP, all we have to do is to define a *second subclass* from the `Array` class that stores real numbers.

Creating a second subclass from `Array` is just as simple as was creating the `IntArray` class -- in fact, a bit simpler, since other subclasses of `Array` will be similar in many ways to `IntArray`. We will now create a subclass called `RealArray` (see Figure 11.18) which will, as its name implies, store an array of floating-point numbers rather than integers (and enforce that *only* real numbers be permitted in the array).

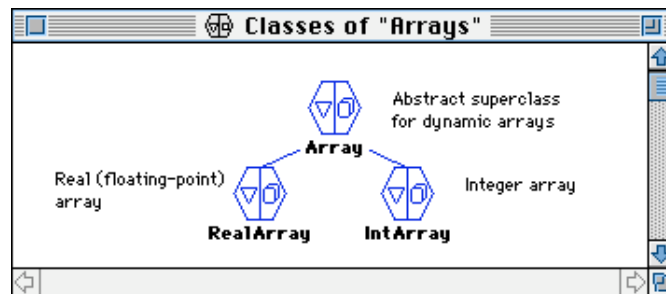


Figure 11.18: The RealArray subclass

The attributes of this subclass, like those of `IntArray`, are inherited from the parent class `Array`. The class methods of `RealArray` are quite similar to those of `IntArray`, except that they deal with *floating-point* elements stored in the `arrayList` attribute. First, the Instance (initialization) method, shown in Figure 11.19. This method creates a list for the `arrayList` attribute, and initializes each member of the list to be a floating-point number equal to 0.0.

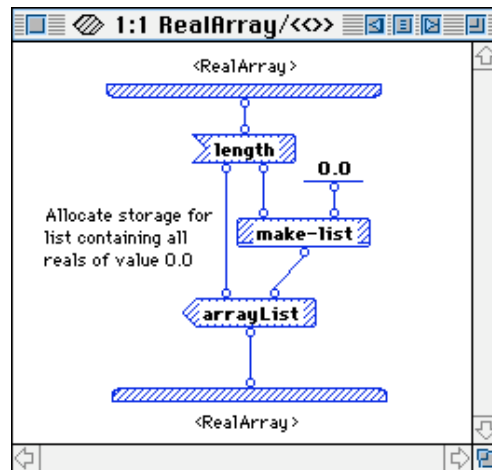


Figure 11.19: Instance or initialization method of the `RealArray` subclass

The `SetElement` class method of `RealArray`, like its counterpart in the `IntArray` subclass, simply checks if the incoming data is of the correct type (in this case, a floating-point value), then sets the `arrayList` member with the superclass' `SetElement` method.

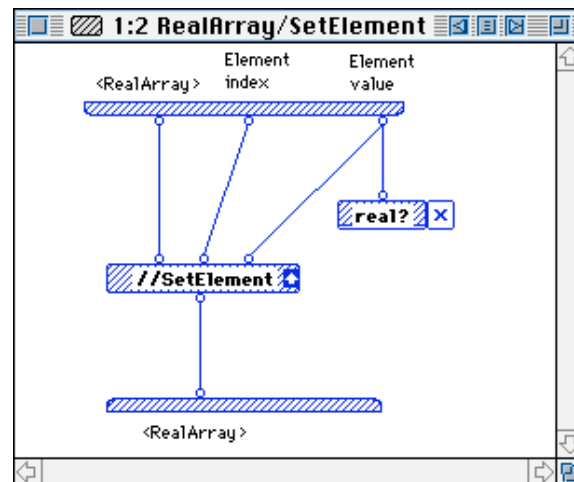


Figure 11.20: `SetElement` method of the `RealArray` subclass

Finally, the `Resize` class method of `RealArray` performs the same actions as does its counterpart in `IntArray`. The sole difference is that the `make-list` primitive in

its first case receives a floating-point value rather than an integer value for initializing the `arrayList` members.

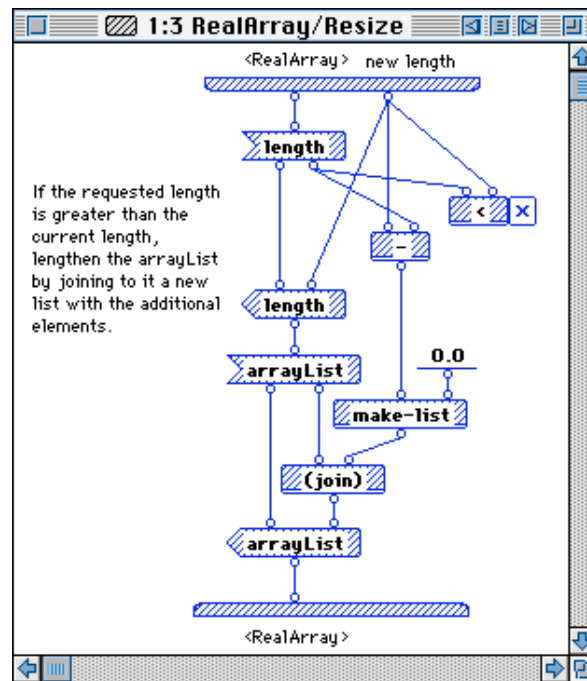


Figure 11.21: First case of the Resize method of the RealArray subclass

What about the **Add** and **Copy** class methods? Once again, polymorphism will ensure that these methods will work correctly. The **Add** and **Copy** methods, which we have placed in the **Array** parent class, will be called appropriately when we use objects of type **RealArray**. In addition, when a **RealArray** class method is called from within **Add** or **Copy** (like **Resize** or the overridden **SetElement** method), the class method from **RealArray** will be correctly executed.

Using the Array Classes

The **Array**, **IntArray** and **RealArray** classes are now complete. With just a little code, we have new data types that store integer or floating-point number arrays. The arrays can be resized, copied to another array or added to another array. That's a lot of function from just a small amount of code. Let's test these classes to see if they are doing what we expect of them.

Create a new universal method in the **Arrays** section called **Test IntArray**. This simple example will show you how to create and use classes in your programs. Complete its code diagram as shown in Figure 11.22. This method creates an object from the **IntArray** class and sets its **length** attribute to 10.

Notice the strange way we're doing this. Instance generators have one input and one output. The output, of course, is an instance of the class -- the object being created. But the *input* may either be left unconnected, or we may feed into this node a *list of values* for the attributes of the newly-created instance. This list will be used to replace the default values we supplied when we defined the class' attributes.

In our Test IntArray method, we've only included one of the class' two attributes in the attribute initialization list -- the **length** attribute. This is safe to do -- you can initialize as many or as few attributes as you want this way.

After the instance generator finishes its work, our custom **IntArray** initialization method is called, which allocates ten elements for the **arrayList**, each with the integer value 0.

The Test IntArray method then sets each element in the array to a value of 1 by entering a loop that repeatedly calls the **SetElement** class method. Next, we create a second **IntArray** object containing only 2 elements, then copy the contents of the first into the second one. Note that since the second **IntArray** is shorter than the first, it will be automatically resized when the **Copy** operation takes place.

Although we have not provided a **show** primitive call here to display the contents of each integer array, their contents can be verified using the debugger. Before execution, set a breakpoint on the **Copy** class method call within the Test IntArray method, then run the Test IntArray method. When execution halts, select the Single Step debugger command, then continue execution. After the **Copy** method executes, use the Value dialog to read the contents of the arrays output from each of the **Copy** method's terminal nodes. You'll find that each integer array now has the same contents -- ten list members each set to 1.

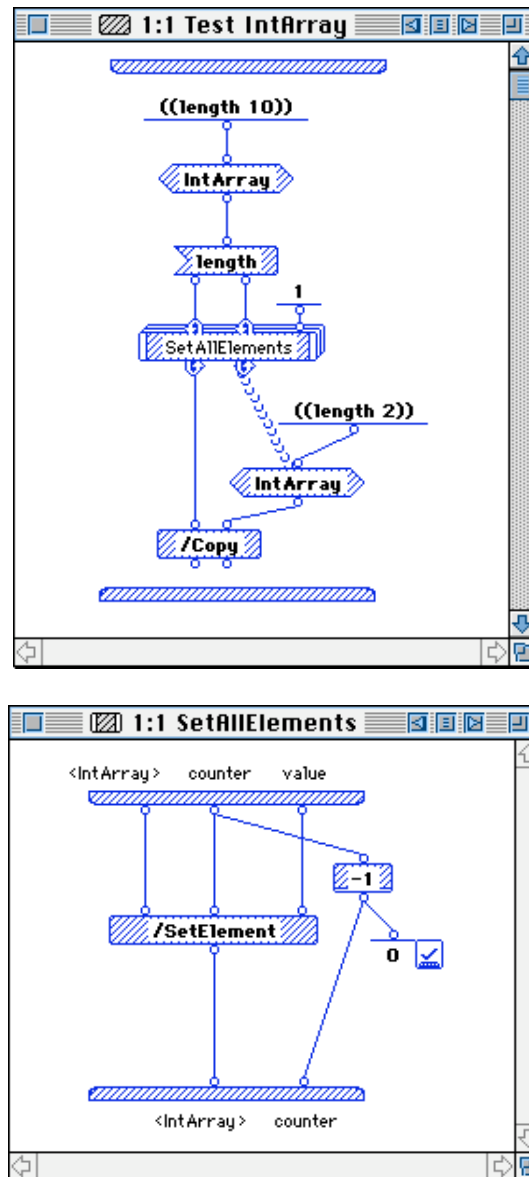


Figure 11.22: The Test IntArray universal method

We now know that the IntArray class functions properly (the RealArray class also works-- trust us, would we lie?). We can now use it just like any built-in Prograph data type. To add integer arrays or floating-point arrays to your programs, all you need to do now is select the Add Sections menu command, then choose the Arrays section. That's all there is to reusing classes you've written in new programs.

Exercise 11.1:

Computer languages like C and C++ include a special data type called a *bit-field*. This is a Boolean-like data type that holds the value (0 or 1) of the bits in a byte, word or long word. That is, a bit-field is a group of bit values combined to form a larger data type holding the values of each bit in a byte or word. For this exercise, create a subclass of **Array** called **BitField** that will hold an array of Boolean values, one Boolean for each bit in a byte. Class methods should allow you to set, clear or test the value of each bit.

Composition

While *subclassing* or *inheritance* is a powerful tool for reusing code, there are applications for which subclassing does *not* work well. For example, look at the relationship between a car and its engine. Suppose we wanted to use two classes -- **Car** and **Engine** -- to represent them. Would we *subclass* **Engine** from **Car**? This would make no sense at all. Remember that subclassing works well in “*is-a*” relationships. Is an **Engine** a *type* of **Car**? Of course not. A **Car** *has an* **Engine** inside it, but an **Engine** is not a *type* of **Car**. The “*has-a*” relationship is better satisfied by the use of *composition* rather than subclassing.

Composition is the placement of one instance of a class inside an instance of another class. This requires a different means of calling class methods as did subclassing. In subclassing, you either called the subclass’ class methods or the superclass’ class methods, but the methods of one class rarely called those of the other or passed data back and forth. You typically called either the parent class’ methods or the subclass’ methods - one or the other. In composition, this all changes. The object containing another class’ instance must frequently call the methods of the contained object, which might return information back. We stated in Chapter 9 that large programs may consist of many objects passing requests for action back and forth; that is, calling each other’s class methods. Composition is a miniature version of such a program. In composition, requests to perform actions (that is, class method calls) and data may be passed back and forth from one object to another within a self-contained system of nested objects.

We will demonstrate composition with a graphical shape class -- the **Line** class. The class will define a line of a given length and orientation that can be moved, rotated, scaled in size, or flipped over. Its length and orientation will be determined by the position of the **Line**’s starting point and endpoint. Each position will be represented with a **Coordinate** class, which stores a two-dimensional position that can be set or moved. Therefore, we will first create a **Coordinate** class. Then we’ll design a **Line** class whose objects will contain within them two instances of **Coordinate**’s. Whenever the **Line** is to be altered in any way, the **Line** must communicate with one or the other **Coordinate** within it.

Create a new section called Shapes and open its Classes window. Create a new class named **Coordinate**. Open the **Coordinate** *attributes* window and define two data elements called **x** and **y** whose default types will be *real* (Figure 11.23). These will hold the x-coordinate and y-coordinate of a location in two-dimensional space.

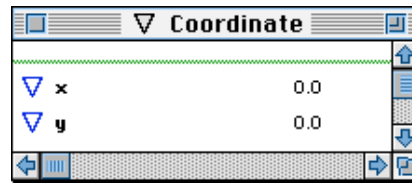


Figure 11.23: Attributes of the Coordinate class

The class methods of **Coordinate** are pretty straightforward. The **GetPosition** method reads the values of the **x** and **y** attributes and returns them as the current position of the **Coordinate**.

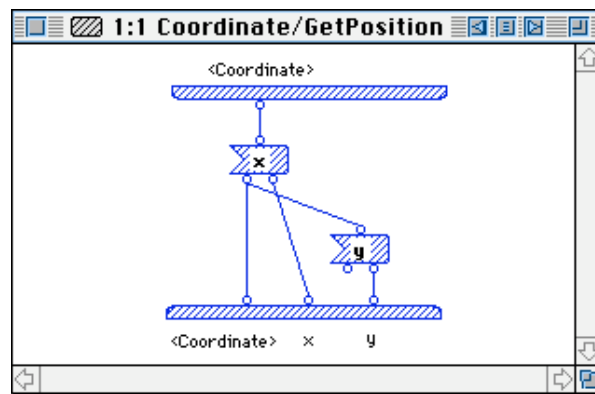


Figure 11.24: GetPosition class method of the Coordinate class

Move does the opposite of **GetPosition**. It sets a new position for the **Coordinate** by putting new values into the **x** and **y** attributes.

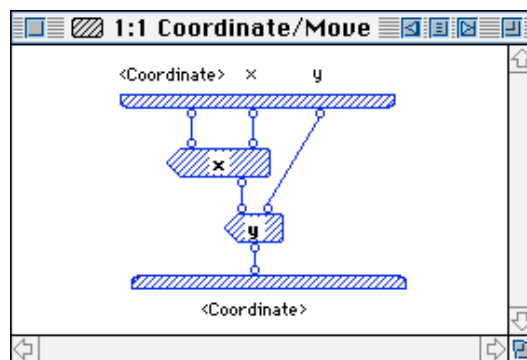


Figure 11.25: Move class method of the Coordinate class

MoveBy is similar to **Move**, but repositions the **Coordinate** *relative* to its current position. It accepts as input the amounts to move the **Coordinate** in the horizontal and vertical directions and adds them to the current values of the **x** and **y** attributes.

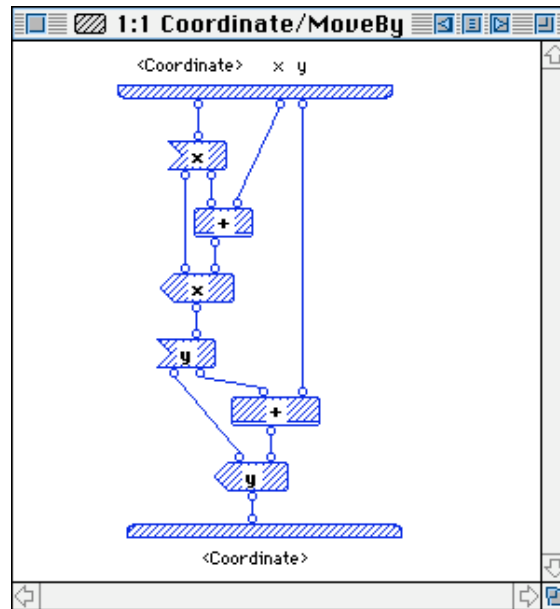
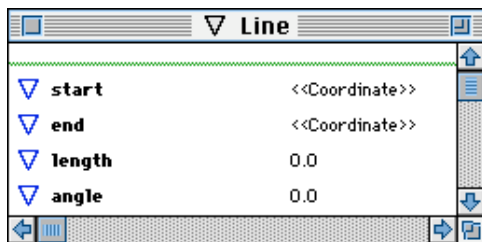


Figure 11.26: MoveBy class method of the Coordinate class

Believe it or not, the **Coordinate** class is finished! All it really needs to do is store a **Coordinate**'s current position or change its position by an absolute or relative amount.

Now let's build upon the **Coordinate** class to create a more complicated data type -- the **Line** class. A line may be described in one of two ways. Either we can just define the position of its start and end, or we can define the line's starting-point location, length and orientation. We will combine both approaches for our **Line** class, providing attributes that will allow for both strategies (Figure 11.27). When we want to define the line in terms of length and angle, we'll access its **length** and **angle** attributes. However, when we want to define the line in terms of its starting and end positions, we will communicate with two instances of the **Coordinate** class embedded within it -- the **start** and **end** attributes.



```
class Coordinate {
    // Class contents
};

class Line {
public:
    // Class functions
private:
    Coordinate start;
    Coordinate end;
    float length;
    float angle;
};
```


Figure 11.27: Attributes of the Line class with their C++ language equivalent

We'd like the **Line** to perform several basic actions. To start, we should be able to do simple tasks such as setting the **Line**'s initial length, position and angle, but let's also add more actions such as moving the **Line** to a new absolute position or a new position relative to its current one, changing its length or rotating the **Line** by a user-defined angle.

Let's look at these operations, one by one. Moving a line means resetting the starting and ending positions of the **Line**. This requires communication with the **Line**'s embedded **Coordinates** -- the **Line** must send messages to the **start** and **end Coordinates**. We will do this in two different ways. To reset the **start** attribute to a new location, we'll call the **start Coordinate**'s **Move** method to do this for us.

For the **end Coordinate**, rather than calling **end**'s **Move** method, we'll use a custom **Set** method for the **Line**'s **length** attribute that will as a course of its own actions also set a new position for the **end Coordinate**. Why are we using such a convoluted technique for setting the **end** position? This technique will ensure that the line endpoint positions will be updated automatically whenever we set a new value for the **length** attribute.

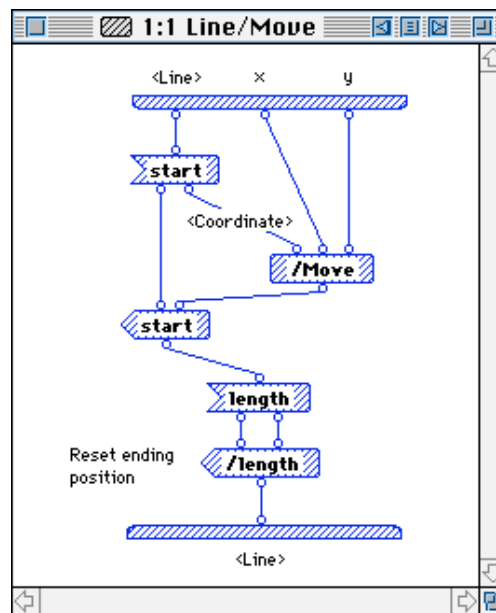


Figure 11.28: Move method of the Line class

The **MoveBy** method, which moves the line to a new position relative to its current position, is quite similar to the **Move** method. The only difference is that we call the **start Coordinate**'s **MoveBy** method rather than the **Coordinate**'s **Move** method.

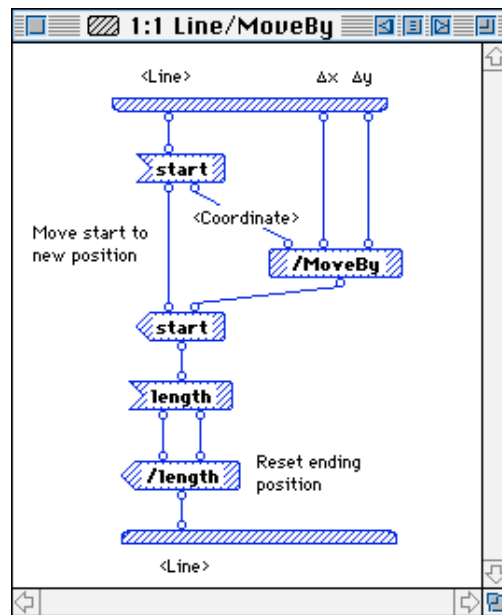


Figure 11.29: MoveBy method of the Line class

What about resetting the orientation of the `Line`? The same method used above for the `length` attribute will work again here. We just add the angle by which the line will be rotated to the current `angle` and call a custom `angle Set` method that will also automatically reset the position of the `end Coordinate` like the custom `length Set` method did (see Figure 11.30).

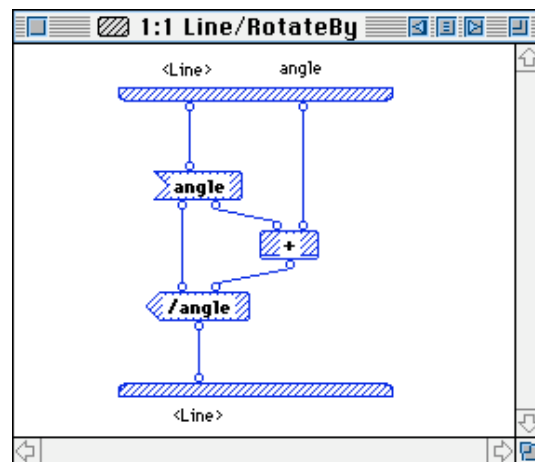


Figure 11.30: RotateBy method of the Line class

Let's return now to the custom *Set* methods for the **length** and **angle** attributes. As shown in Figure 11.31, the **length** attribute is set by first calling the built-in **length** Set operation, then using the current values of the **length** and **angle** attributes to calculate the new position of the **Line's end Coordinate**. This is done by feeding the

length and angle values, as well as the start and end Coordinates to a universal method called VectorToPoints, which outputs the new end Coordinate value.

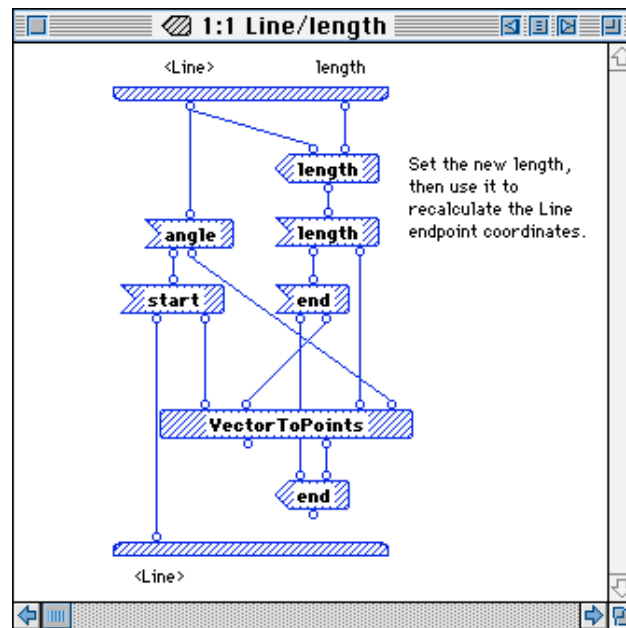


Figure 11.31: The overridden length Set method of the Line class

The VectorToPoints universal method (see Figure 11.32) simply calls two local methods to reset the end Coordinate's x and y attributes. Each local method, depicted in Figure 11.33, must convert the length and angle of the line back from *polar* (length-angle) coordinates to *Cartesian* (x-y) coordinates. In doing so, the method resets the endpoint x-y position. The mathematical formulae we use for these calculations require angular inputs in radians. We can reuse a method we wrote earlier to help us -- the DegreesToRadian method.

In the SetEndX local method, we convert the angle attribute from degrees to radians, then calculate its cosine. We multiply this cosine times the Line's length attribute to get the change in position along the x-axis from the line's start to its end. We add this to the x position of the line's start to get the x position of the end Coordinate.

In the SetEndY local method, we carry out precisely the same steps, using the sine of the angle instead to get the change in position along the y-axis from the line's start to its end. We add this to the y position of the line's start to get the y position of the end Coordinate.

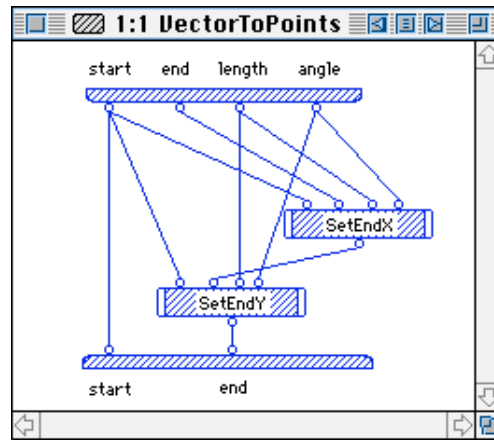


Figure 11.32: The VectorToPoints universal method

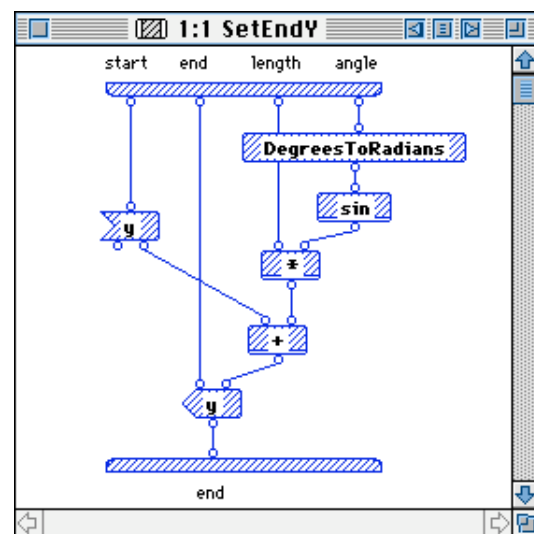
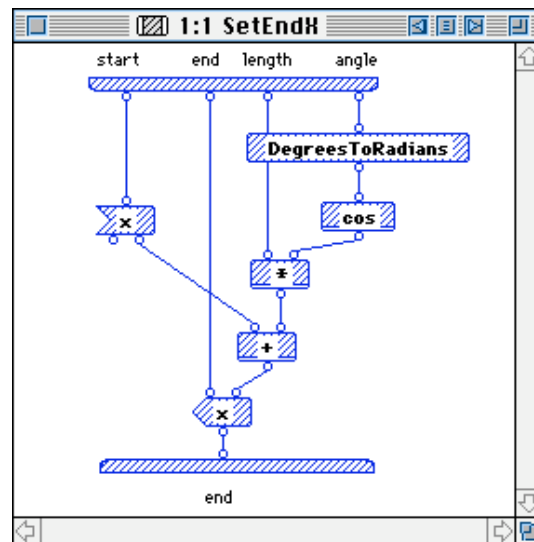


Figure 11.33: The SetEndX and SetEndY local methods

The custom *Set* method for the **angle** attribute is depicted in Figure 11.34. The **angle** attribute is set with the built-in **angle Set** operation. Next, as we did in our **length Set** method, the current values of **length** and **angle** are used to calculate the new position of the Line's **end Coordinate** via the **VectorsToPoints** universal method.

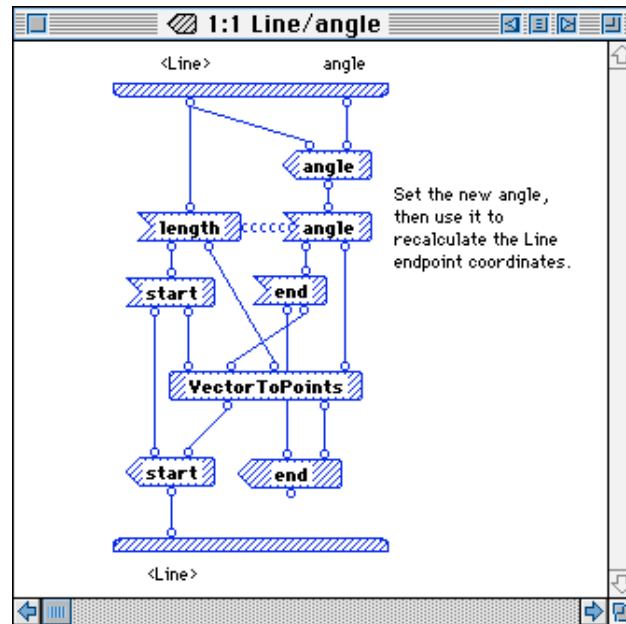


Figure 11.34: The custom angle Set method of the Line class

This completes the **Line** class. Our **Line** can be placed anywhere, resized and rotated. For all its simplicity, the **Line** class does a lot. And it can be reused in any graphics program we choose to write that needs to represent (or draw) lines. The **Coordinate** and **Line** classes could form the start of a *device-independent object-oriented drawing system*. What does this mean? “Device independent” means that we define the position of a graphics shape (such as our **Line**) in abstract terms, and not solely by the pixels on our computer’s monitor. This is important because hardware such as a computer monitor is always subject to change. If our shape classes have no direct dependence upon the specifics of a particular monitor, we can use them for *any* monitor. Our **Coordinate** and **Line** classes use only abstract x and y coordinates in space, not specific positions on the monitor screen. An “object-oriented” drawing system uses graphics *objects* instead of collections of pixels or dots on the screen. Again, our classes meet this definition. The **Line** class is defined by its ends, length and orientation, not by each individual point along the line. This model of drawing is the same as used by illustration programs. You create a shape such as a line, then move it, rotate it, etc., since the line is stored as a *definition of a line*, not as points falling on the line. What we’ve done is to write basic utility classes for a drawing package -- a way to define the placement of a graphics object (**Coordinate** class) and one drawing shape (**Line** class). These simple shape classes can be built upon to create other shapes such as *rectangles* or *irregularly-shaped polygons*. The **Coordinate** class can be included in a *curve* or *circle* as well.

Exercise 11.2:

Create a **Rectangle** class that is defined by the position of its top-left corner and bottom-right corner, as well as its height, width and orientation. Allow the **Rectangle** to be resized, moved or rotated.

Summary

Inheritance and composition are our main techniques for extending the usefulness of our classes and to create new classes by reusing previously-written code. In this chapter, we examined how to take advantage of these techniques with Prograph.

- **Inheritance (subclassing)** is directly supported in Prograph by creating a new class, then making a datalink between the desired parent class and the subclass. The attributes of the subclass are given a slightly different icon to mark them as inherited attributes.
- **Abstract superclasses** are a special type of superclass that is never used to create objects. Its sole purpose is for subclassing. Its child classes may be used to create objects.
- **Instance methods**, discussed in the previous chapter, may also be used to initialize attributes. In this chapter, we used them when creating objects from the **Array** class to set the lengths of the class' list attribute, then allocate memory for the contents of the list.
- When a class method in the parent class does not perform the exact action we wish a subclass to perform, we may **override** the parent class method with a new method in the subclass. The overridden method may in turn call a **superclass method** to reuse the code in the parent class' method that we do wish to use in the child class method.
- **Composition** is accomplished in Prograph by setting the value of an attribute of one object to be an instance of another class. We make use of the embedded object by calling its class methods.