

12. Using the Full Power of OOP

Overview

The process of inheritance that we examined in the last chapter affords us with the ability to build new classes using features of other, already coded and tested, classes. This mechanism would not be very useful if we always had to be aware of *which* class we were sending requests to -- a parent class or its subclass. As an example, let's say we fed an instance of either a class such as the **IntArray** or a subclass of it called **SortedIntArray** into a method (note that in this case, **IntArray** is itself a parent class as well as the subclass of the more general **Array**). If we wanted to execute a class method of whichever instance arrived as that input, we would have a lot of difficulty if we had to actually *identify* the incoming object. How would we know if it was an object of type **IntArray** or **SortedIntArray**? There'd be no easy way to know. Fortunately, *polymorphism* removes the need to know this. In this chapter, we'll examine how polymorphism both simplifies our programming task and enables more powerful uses of classes. We'll also take a brief detour into the use of *class attributes*, data shared among several objects of a single class.

Polymorphism

One of the advantages of using similar class method names within similar classes is that we can take advantage of *polymorphism* -- the ability to send the same request to different objects without the need to know exactly *which* object is satisfying the request. Let's see how this works with a concrete example.

Add a **ShowPosition** class method to each of the shape classes we've developed so far: **Coordinate**, **Line** and **Rectangle** (from Exercise 11.2). Your **Rectangle** class may be slightly different from ours, which uses composition and includes four **Line** objects to represent the four sides of the **Rectangle**. You may have to modify the **Rectangle/ShowPosition** method to match your own **Rectangle** class design.

The **Coordinate** class' **ShowPosition** method (Figure 12.1) prints the values of the **x** and **y** attributes. The corresponding method for the **Line** class (Figure 12.2) prints the horizontal and vertical positions of its **start** and **end** points. Finally, the **ShowPosition** method of the **Rectangle** class (Figure 12.3) prints the **start** and **end** points of each of its constituent **Lines**.

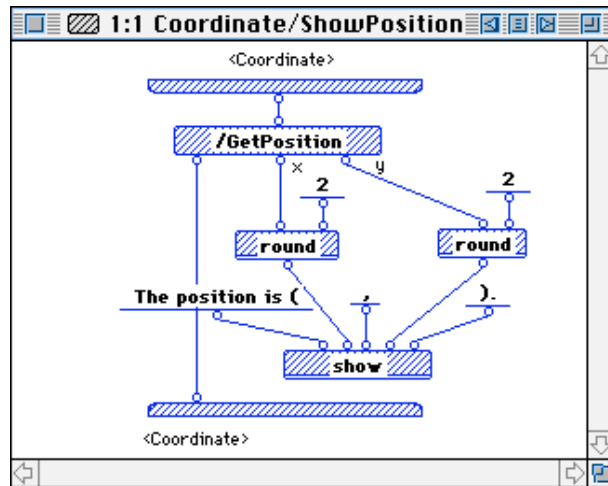


Figure 12.1: ShowPosition method of the Coordinate class

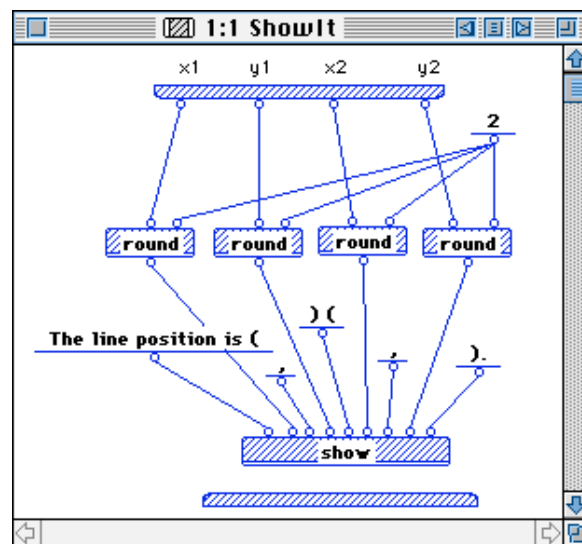
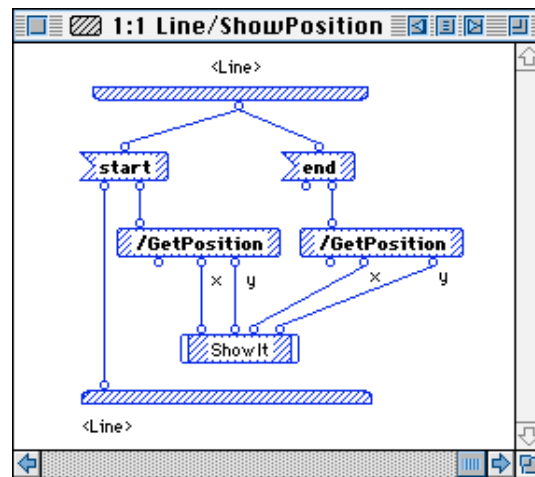


Figure 12.2: ShowPosition method of the Line class

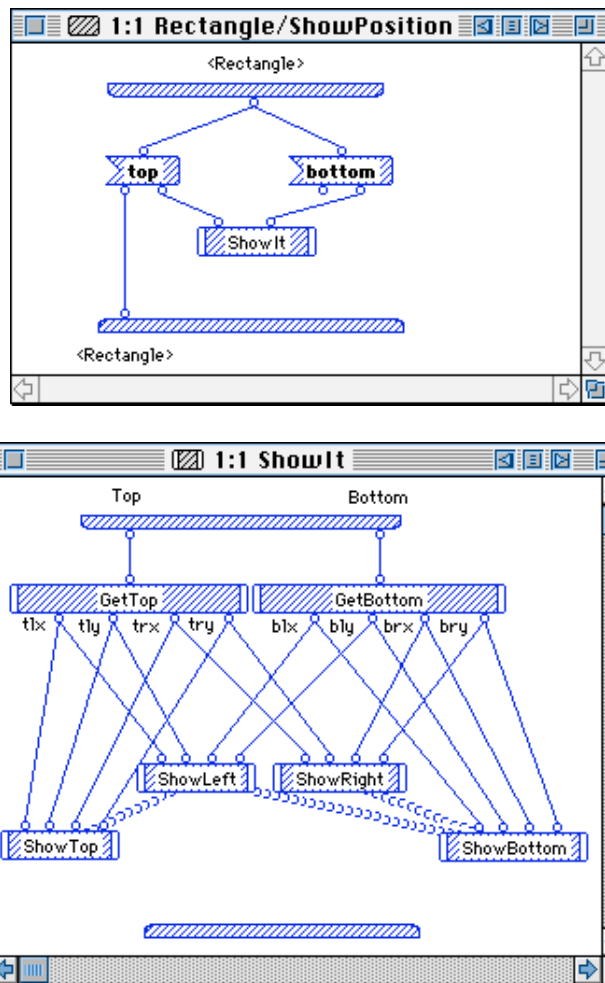


Figure 12.3: ShowPosition method of the Rectangle class

These three class methods all perform equivalent actions. They print out the current position of the **Coordinate**, the **Line** starting-point and endpoint, and the **Rectangle** upper-left and lower-right corners. We should be able to send the same **ShowPosition** request to either a **Coordinate**, **Line** or **Rectangle** object and have the position of the appropriate shape displayed on the screen. Let's try it! Create a persistent called **shapeList** (see Figure 12.4) and make it of type *list*. This list will serve to hold a collection of different shape objects.

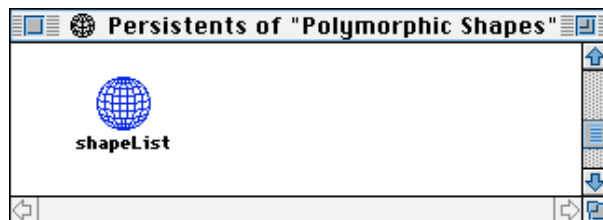
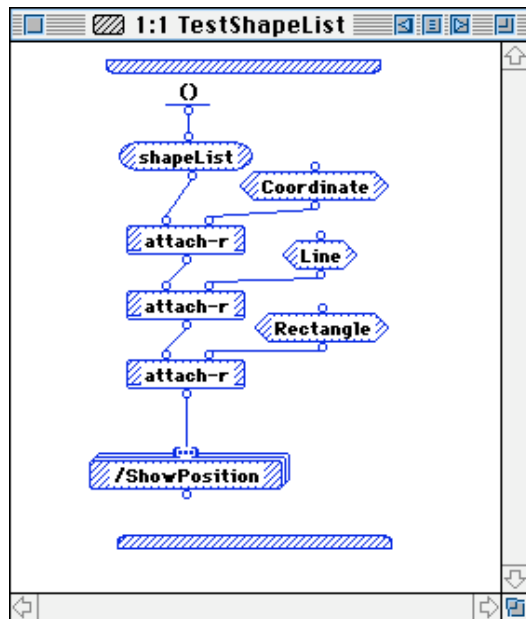


Figure 12.4: shapeList Persistent for holding list of objects created from shape classes (Coordinate, Line and Rectangle)

Now create a universal method called TestShapeList and complete its code case window as shown in Figure 12.5.



```
Coordinate point;
Line      line;
Rectangle rect;

// Hypothetical
// parameterized
// List class containing
// Shape objects

List<Shape>  *shpLst;

void TestShapeList( void )
{
    shpLst = new List<Shape>;
    shpLst->Add( point );
    shpLst->Add( line );
    shpLst->Add( rect );
    for (short i = 0; i <
        shpLst->Length(); i++) {
        // Overloaded [] operator
        shpLst[i]-
        >ShowPosition();
    }
}
```

Figure 12.5: TestShapeList universal method and a possible C++ counterpart

The TestShapeList method is fairly simple. After initializing the shapeList persistent to an empty list, it attaches a **Coordinate**, **Line** and **Rectangle** to the persistent's list of **Shape** objects. Then, using a list multiplex, sends a **ShowPosition** request to each shape object in its list, one after another. The TestShapeList method does not know what shapes are stored in the persistent or in what order they are in. All it knows is that it has sent the same **ShowPosition** request to each object in the persistent's list. It is up to those objects themselves to correctly respond to that request.

When we execute the TestShapeList universal method, we are presented with three Show dialogs that display the position of the **Coordinate**, **Line** and **Rectangle** in that list in turn. These outputs are shown in Figures 12.6-12.8 (only the output for the top line of the **Rectangle** is shown here). Each object in the list, upon receiving the

ShowPosition request, does in fact display the position of the shape that the particular object represents in a manner specific to that shape.

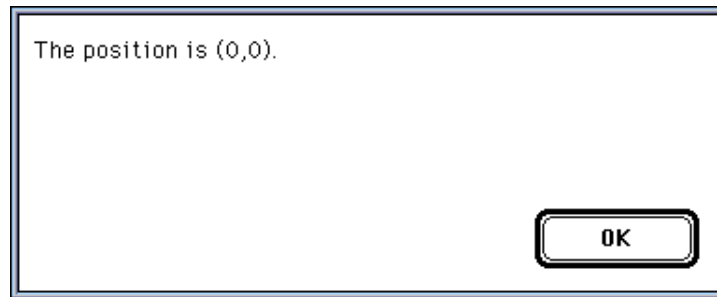


Figure 12.6: Output of the TestShapeList method for Coordinate object

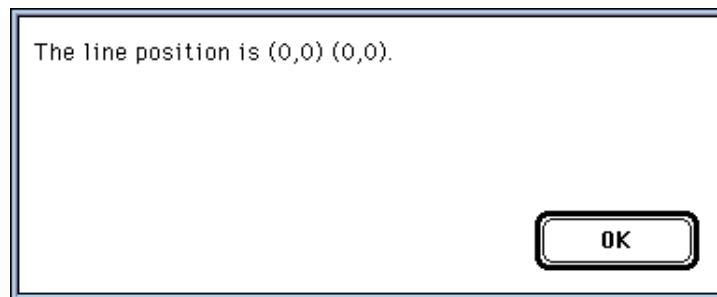


Figure 12.7: Output of the TestShapeList method for Line object

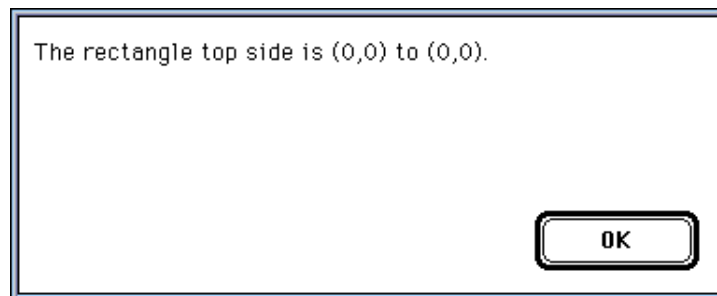


Figure 12.8: First show primitive output display for Rectangle object

Prograph did not need to know that the list contained a Coordinate, Line and Rectangle. to handle each ShowPosition request. All that mattered is that each class, whatever it was, had the ability to accept and process a ShowPosition request. We no longer have to explicitly send a specific request to one object (Coordinate/ShowPosition), then another specific request to the next object (Line/ShowPosition), etc. We can treat the list of shapes just like any other list and iterate the *same* request (/ShowPosition) to each its members *in the same way* using a list multiplex.

The ability to send identical requests to all members of a list of objects opens up a lot of possibilities. As we just saw, it could be used in lists of shape objects in a drawing program. For example, to update a drawing program's display window, we need only send a **Draw** request to each shape in its list of shape objects via a list multiplex. To update the salary of all employees in a company, no matter if they're factory workers, executives or janitors, we can send an identical **GiveRaise** request to a list of objects subclassed from an **Employee** class. Polymorphism provides us both with a way to reduce the number of different method names we need to use with different classes and with a powerful way to use lists and objects together.

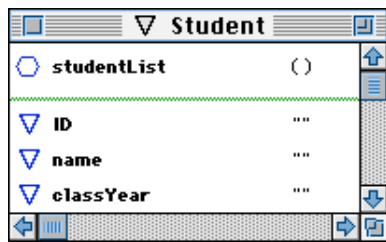
Shared Class Attributes

In Chapter 9, we mentioned that attributes come in two types -- *instance* attributes and *class* attributes. So far, we've used only *instance* attributes, whose value differs in each object created from that class type. This section will focus upon *class attributes*. These are attributes that are *shared* by *all objects* created from that class. In C++, *static* class members serve this function.

Class attributes allow us to do two powerful tasks. First of all, we can set up an attribute whose value is the same for all objects of that class type. This class attribute serves in a sense as a *constant* shared by all of the objects. So, for example, we could place a commonly-used number like **salesTaxRate** in every object of type **PurchasedGoods**. Any method that needed to access the sales tax rate would read the value of the **salesTaxRate** class attribute. If the value of **salesTaxRate** were to be changed, it would automatically change *for every object of that class*.

The second use of class attributes is the inclusion of a *list of objects*. Let's say we wanted to make a list of students taking a college course. Instead of storing the student list in a persistent, we could store it *within the class itself* by using a class attribute. The list class attribute will act just like a persistent (even retaining its previous value when a program is rerun, just like a persistent) but it is encapsulated into the class.

Create a new program, and make a new section called **StudentList** and a class called **Student**. Open the **Student** class attributes window, and complete it as shown below. There are four attributes. Three of these are instance attributes, containing the **Student's** ID number, name and class year. The fourth is a *class attribute* called **studentList**. The **studentList** attribute will contain a list of *all of the objects* we create from the **Student** class. At present, it is empty.



```
class Student {
static List<Student> fStudentList;
public:
    // Class methods
private:
    short fID;
    char fName[ 256 ];
    short fClassYear;
};
```

Figure 12.9: Class attributes and instance attributes of the Student class with the C++ equivalent using a static parameterized List class

The first class method of the Student class is FindStudent (Figure 12.10). It accepts an index into the `studentList`, and returns that corresponding Student.

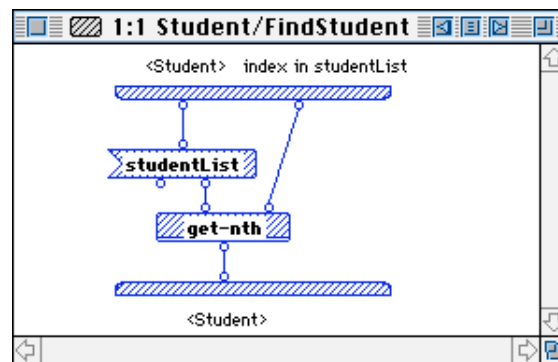


Figure 12.10: FindStudent method of the Student class

The `ClearStudentList` method, shown in Figure 12.11, removes the current contents of the `studentList` by setting the `studentList` to an empty list. You may have noticed that we do not feed an instance of the Student class as an input into this class method. This is completely different from every other class method we've written up to now. Why are we doing this? We'll explain this shortly.

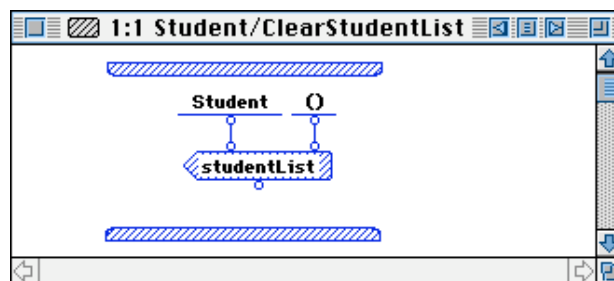


Figure 12.11: ClearStudentList method of the Student class

The `ShowStudent` method prints out the values of all of the instance attributes of the `Student` object receiving this message.

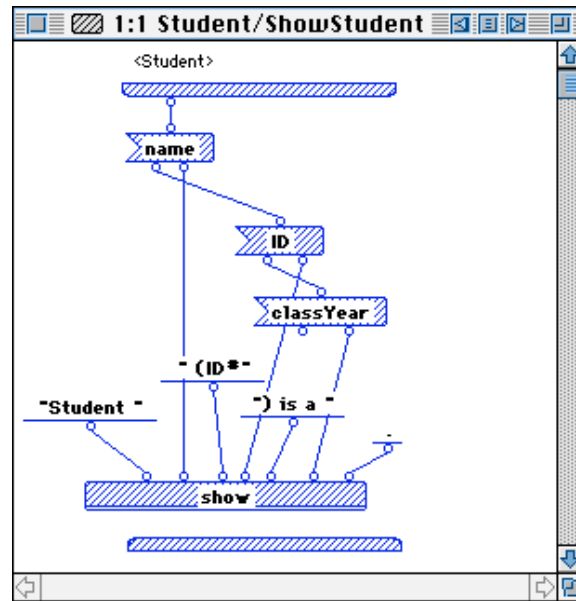


Figure 12.12: ShowStudent method of the Student class

The `ShowAllStudents` class method takes advantage of the `ShowStudent` method (see Figure 12.13). `ShowAllStudents` reads the `studentList` attribute of a `Student` object, then calls the `ShowStudent` method repeatedly with a list multiplex. The result is that the attributes of each `Student` object is displayed.

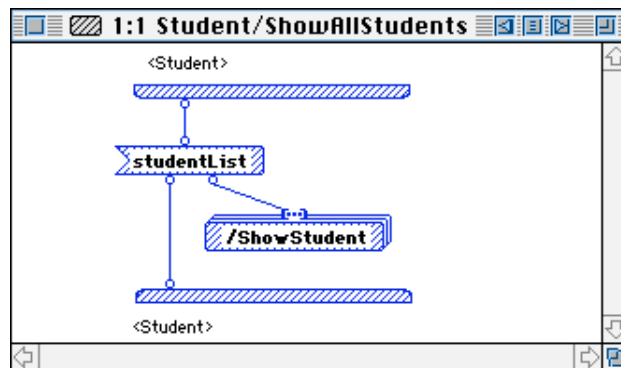


Figure 12.13: ShowAllStudents method of the Student class

The last class method is the instance method (see Figure 12.14). It first initializes the attributes of a newly-generated instance of **Student**, then inserts this new **Student** onto the end of the **studentList**.

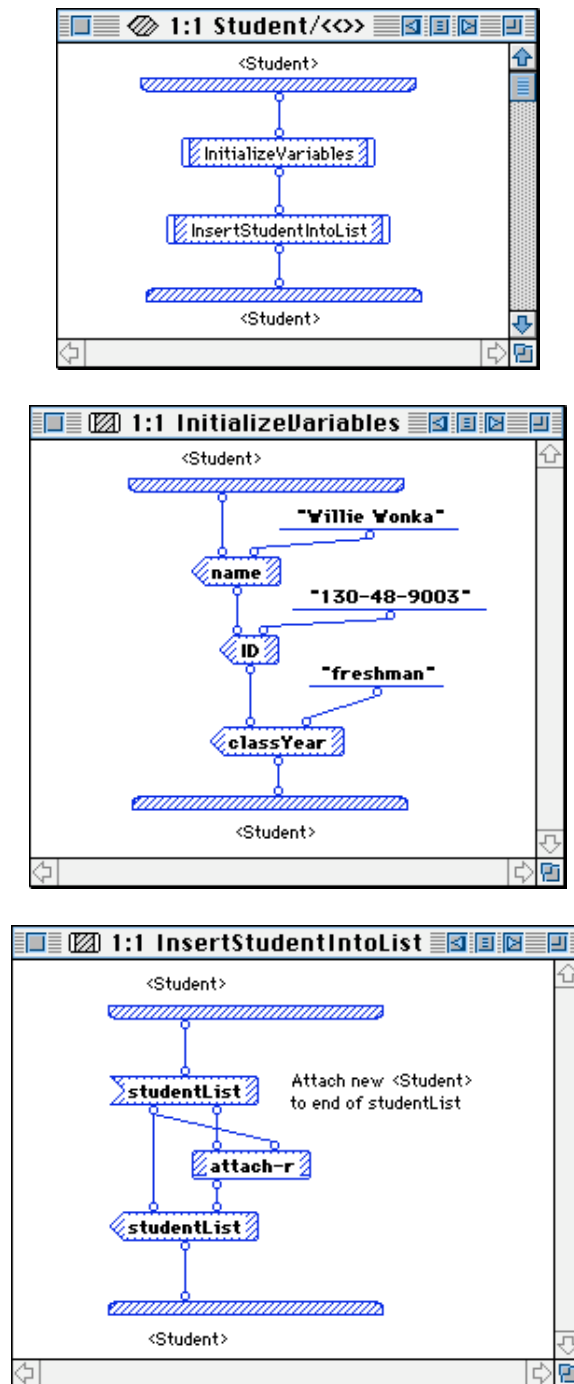


Figure 12.14: Instance method of the Student class

Finally, we come to the universal method that will test the `Student` class -- `TestStudentList`. This method clears the `studentList`, creates two objects of type `Student`, then uses the `FindStudent` class method to access the second `Student` in the `studentList`. It passes the second `Student` to the `TestStudent` local method, which resets its instance attributes to new values. We do this so we can tell apart the two students

when we next display their attributes by getting the `studentList` and sending a `ShowStudent` request to each `Student` in the list.

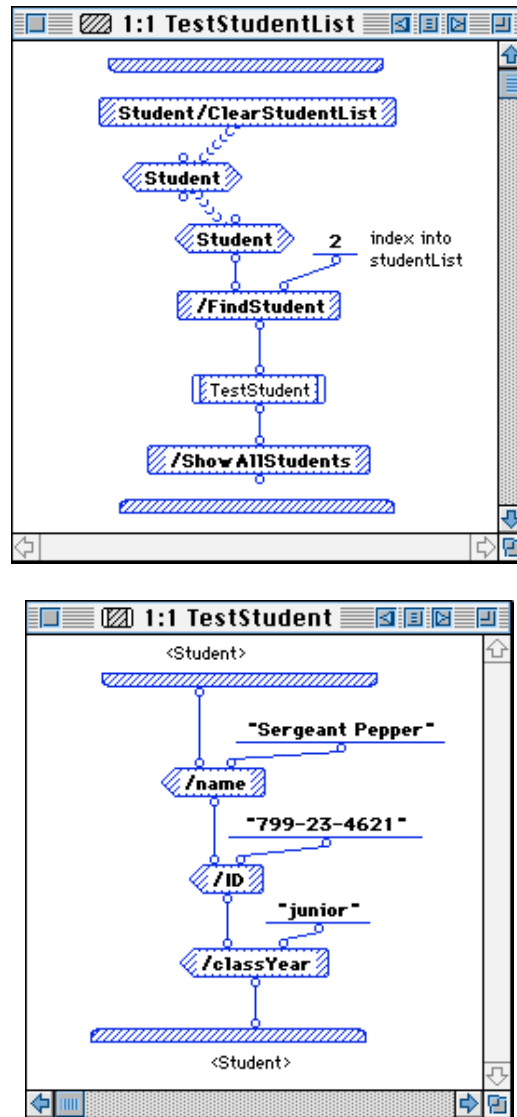


Figure 12.15: TestStudentList universal method

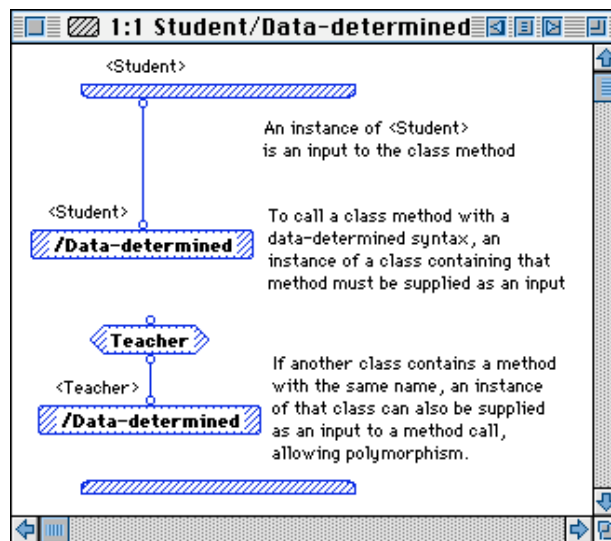
Did you notice that the `TestStudentList` method called the `ClearStudentList` class method (whose code was shown in Figure 12.11) without feeding it an instance of `Student` for an input? Now we can explain why `ClearStudentList` was written the way it was.

Class methods may be called in any one of three different ways (see Figure 12.16): The first is the *data-determined* class method call, with the syntax “/MethodName”, where the first input to the method must be an instance of a class containing a class method named `MethodName`. This is the technique we’ve used so far in Chapters 10 through 12. This technique allows for polymorphism, since more than one

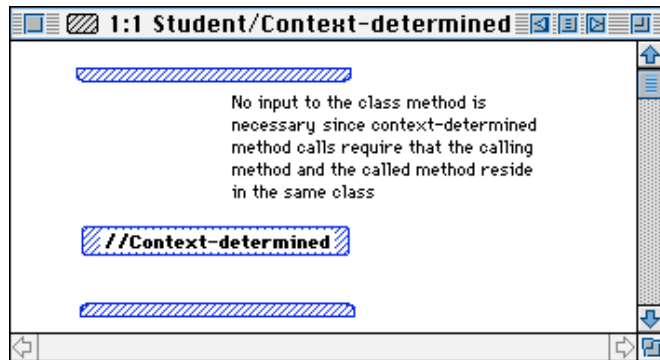
type of object can be used as an input to the method call. The method that is actually executed is determined by which instance arrives as that input. The second way to call a class method is a *context-determined* method call, which has the syntax “//MethodName”. This may be used when one class method calls another class method. Here, the called method is assumed by Prograph to be in the same class as the method that is calling it. Therefore, we don’t need to supply an input instance to the called method. The third way of calling a class method is what we used for the **ClearStudentList** method. it, too, does not require the input of an instance of a class into the called method. This is why the **ClearStudentList** method of Figure 12.11 has no inputs, and why the **TestStudentList** method (see Figure 12.15) feeds no objects into **ClearStudentList** when calling it. This is an *explicit* method call, in which the name of the class containing the class method to be executed is *explicitly* stated in the form “Class/MethodName”.

If we had not used an explicit method call in the **ClearStudentList** method, then a third instance of **Student** would have to have been generated in the **TestStudentList** method for the sole purpose of using it as input to **ClearStudentList**. This is waste of both memory space and execution time. Remember that **studentList** is a *class attribute* - any time the **ClearStudentList** class method is called, this shared attribute will be cleared for *all* existing or nonexistent **Student** objects. Even if the **ClearStudentList** method is not called by an *existing* instance of the class, it will still clear the shared attribute in memory. With an explicit method call, an extra instance of **Student** need not be generated simply to call **ClearStudentList**.

Data-determined class method call



Context-determined
class method call



Explicit
class method call

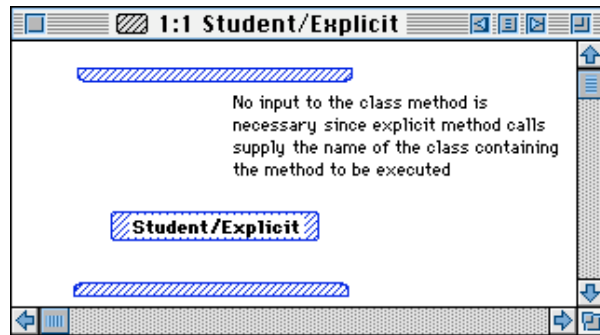
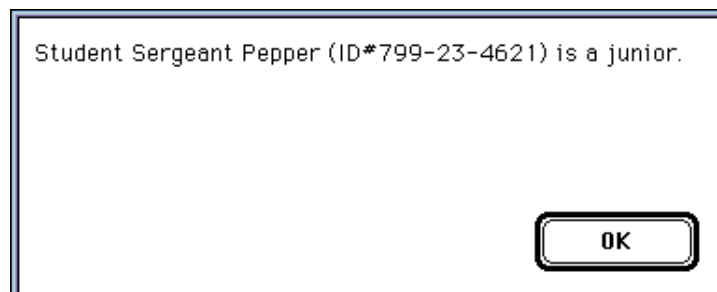
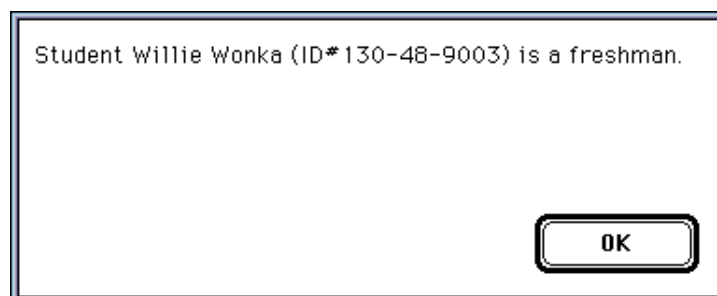


Figure 12.16: The three ways to call a class method

What is the outcome of the TestStudentList method? It displays three Show dialogs, depicted in Figure 12.17, that tell us the settings of the instance attributes of each Student in the studentList. The first and third Students will have the default settings for its instance attributes, while the second Student will have the instance attribute values we set for it in the *TestStudent* local method of TestStudentList.



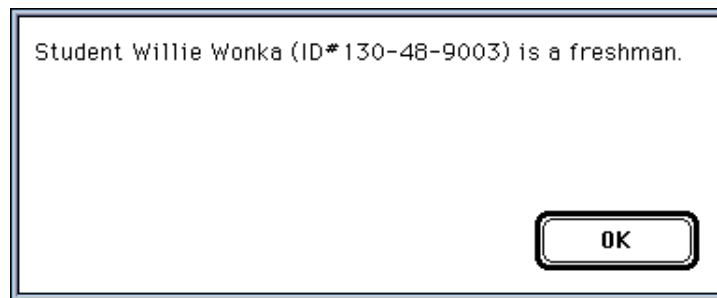


Figure 12.17: Output of the TestStudentList universal method



Warning!

Class attributes behave in many ways like persistents embedded within an object. When a program is run in the Prograph *interpreter*, class attributes *retain* their value between program executions much like a persistent. If you use a class attribute to contain a list, as we did in the Student List example program, that list will *maintain* all of its items when the program is run again; you must *explicitly clear the list* either by: (1) editing the class before executing the program again or (2) calling a class method to clear the class attribute's list when the program starts up. If such a list is used in *compiled* programs, we would have to use the **save** and **load** primitives to explicitly save the class attribute's value to disk then restore it the next time the program is run. Otherwise, the class attribute will be initialized to an empty list each time the program is run.

Exercise 12.1:

This is an optional, more difficult exercise. Write a small OOP program that includes the classes **Student**, **Instructor**, **Course** and **Schedule**, using subclassing to design the **Student** and **Instructor** classes. The focus of this exercise is on message passing between classes, so for example if a **Student** needs to enroll in a course, the **Student** should request the **Instructor** for permission to do so. The **Instructor** will decide whether or not it will be permitted by sending a message to the **Course** to check if the **Student** has completed the prerequisites of the **Course**. Include a universal method to test the program. This method should build a course schedule containing a list of courses offered, the course instructors and the time each course is taught.

Summary

Object-oriented programming offers many constructs that foster careful code design and writing. This chapter wrapped up the final two techniques that are available in Prograph's implementation of OOP.

- ***Polymorphism*** allows the passing of a single message to many types of objects. Each object will carry out the actions appropriate for their class. The programmer does not need to keep track of the identity of each of these objects. this enables the use of objects in lists, such as graphic shapes that can all be moved, rotated or drawn using uniform messages sent to each item in the list.
- Rather than store a copy of common information within every object of a single class as individual *instance* attributes, ***class attributes*** allow the sharing of a single stored data attribute.
- Class methods may be invoked with three different methods, each with its own unique syntax: ***data-determined*** (/MethodName), in which the object containing the method to be executed is supplied as an input to the method call, ***context-determined*** (/MethodName), where the calling method and the called method are in the same class, and ***explicit*** (Class/MethodName), where the class containing the method is stated in the method call.

In Chapters 9 through 12, we discussed object-oriented programming in great depth. You as a programmer are under no obligation to use OOP with Prograph. We've already seen that we can write structured procedural programs in Prograph just as easily. However, OOP allows us to reuse code more than does procedural programming. More importantly, OOP allows us to understand the class system that comes bundled with Prograph -- the *Prograph Application Builder Classes (ABCs)*. These classes are the focus of the upcoming Chapters 14-17, but first, we'll examine some practical applications of object-oriented programming outside of the ABCs.