

10. Applying OOP with Prograph -- Classes

Overview

Now that we've reviewed the background theory of object-oriented programming, it's time to put OOP into practice with Prograph. In this chapter, we'll show you how to create some simple classes and use objects in Prograph programs. The next few chapters will build upon this knowledge to construct more complex classes and take full advantage of OOP. The concepts discussed in this section will prove to be very important when we outline how to add a standardized user interface to our programs using the *Application Builder Editor and Classes* in Chapters 14 through 17. Luckily, Prograph makes object-oriented programming easy to use.

Creating Classes with Prograph

Let's start with an example that will demonstrate just how easy it is to construct and use classes -- a version of the **Car** class that we discussed throughout Chapter 9. This example, although not very practical for the programs you'll most likely want to write, will demonstrate many of the principles of using OOP with Prograph. We'll use a subset of the possible properties and actions that a real car could have, as shown in the **Car** class diagram of Figure 10.1.

class Car	
speed direction gear	Ignition Shift Gears Accelerate Brake
engine tires owner	Stop Turn

Figure 10.1: Attributes and class methods of the Car class

Create a new project named Car project, and a section named Car. Open the Car section's Classes window, then create a new program element. Name this element **Car**, as shown in Figure 10.2. Congratulations! You've just created your first *class* -- a definition of how an *object* will be formed.

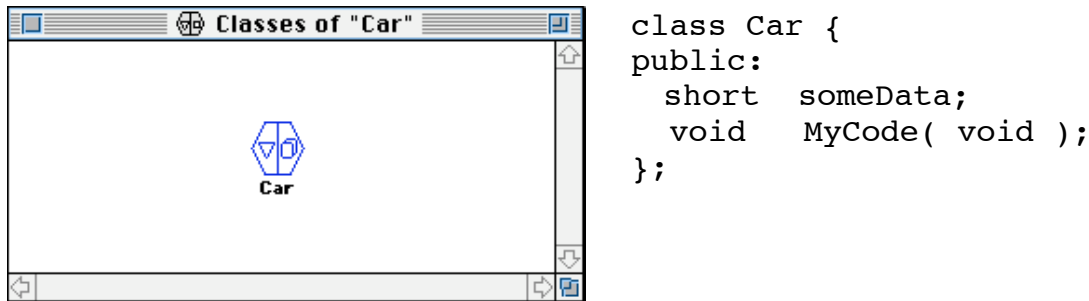


Figure 10.2: The divided icon of the Car class reflects the class' containment of both data and code. The equivalent C++ code is also shown.

Notice that the *class* icon is much different than the other Prograph program element icons we've seen so far. The *class* icon is divided in half. This reflects the dual nature of classes -- they contain both *data* and *code*. The left side of the icon has a triangular symbol that represents the class' *attributes* (its data), while the stacked rectangular symbol on the right side of the class icon represents the *class methods* (its code). To construct a fully-functioning class from which practical objects may be made, we must define both the class' attributes and class methods.

If you examine the C++ language class definition in Figure 10.2, you'll find that it's much less obvious where data and code are being declared -- the declarations look similar at first, and both code and data are declared in the same place in the class definition. This requires the programmer to examine the source code very carefully. In Prograph, we define attributes and class methods in separate windows, which reinforces their separate roles within the class. Prograph's visual nature helps us grasp class definitions easier than does textual programming.

Attributes

We'll start by defining the attributes of the class. Click on the left-hand side of the **Car** class icon. A window entitled **Car** will open (see Figure 10.3), displaying a triangular *attributes* symbol (like on the left side of the class icon) in its title bar. The data elements of the **Car** class will be defined here.

Note that the attributes window has a horizontal line running across it. This line subdivides the window into two attribute regions. The bottom region will contain its *instance attributes* -- those attributes whose values may be different from one object of this class to another. The top region will be used for its *class attributes* -- data elements whose single value is *shared* by all objects of this class. Class attributes are the equivalent of C++'s static class members. We'll return to a discussion of class attributes later in this chapter. For the moment, let's focus on the **Car** class' instance attributes.

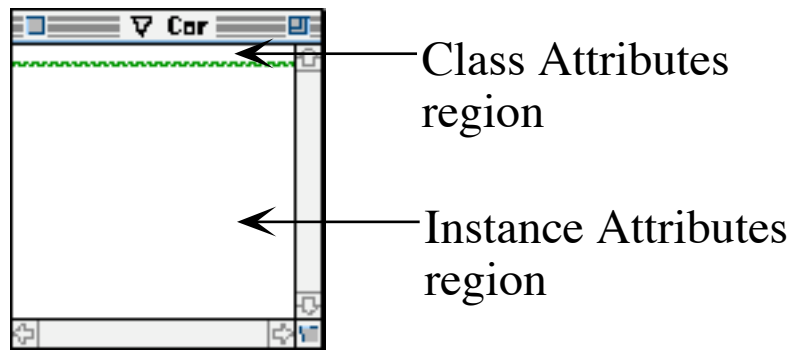


Figure 10.3: Attribute window for the Car class

Click once, while holding down the option key, inside the instance attribute region of the window (below the line in it). A triangular symbol will appear that denotes an attribute -- a data element of this class. Give the attribute the name **speed**.

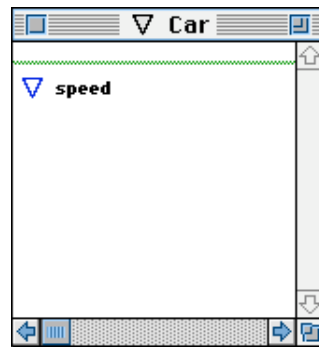
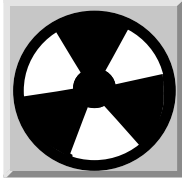


Figure 10.4: Creating a new attribute for the Car class

This attribute will hold the current speed of the **Car** in miles per hour, so **speed** should be a *real* variable. Attributes are created with a default type of *NULL*, so we must define the attribute's data type by opening its Change Value window. Choose *real* from the pop-up of data types. If you widen the **Car** attribute window, you'll see the value of the **speed** attribute displayed in it (see Figure 10.5).



Even though we are discussing the definition of an attribute's data type, there is actually no way to ensure that the attribute remains that data type! Prograph is an *untyped* programming language. That is, its variables, persistents and class attributes can hold *any data type* during program execution, and can even *change* data types! So when we talk about defining the data type of an attribute, we are really only specifying the attribute's *default* data type and value, which could change later in the program. If we want the attribute to always be a particular fixed data type, we must perform *data type checking* whenever we set a new value for the attribute, and only set the new value if it's the correct data type. Otherwise, we may have to change the data type to the correct type (*typecast* it) before setting a new data value.

In C++, you must initialize class attributes in special methods called *constructors*; in Prograph, such initialization methods are needed less often since you can initialize data within the class definition itself. When an object is created from this class, its data will be initialized to the values you have set, automatically.

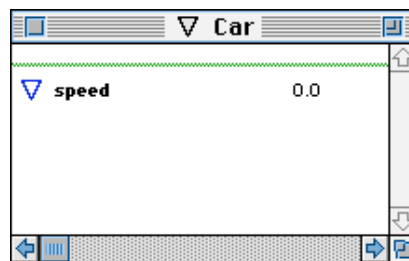


Figure 10.5: Displaying the value of the speed attribute

Define the remaining attributes of the **Car** class as shown in Figure 10.6, using the appropriate data type (real, integer or string) for each attribute's default value.

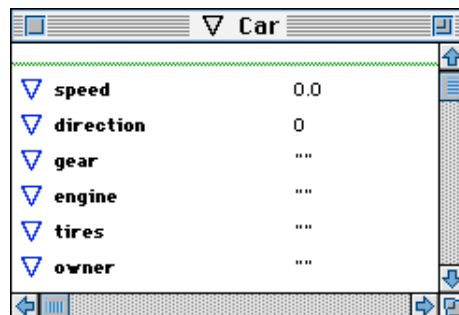


Figure 10.6: The attributes of the Car class

The attributes of our minimal **Car** class have now been fully defined. These include attributes that describe the *traits or parts* of the **Car** -- the **tires** and **engine** that may make one instance (*object*) of the class unique from all other instances. Likewise, the **owner** of the **Car** may differ for each instance. The attributes also contain *state variables* that define the *current status* of the **Car**, such as its **speed**, **direction**, and current **gear**. Each **Car** possesses these attributes so that we can use several **Cars** in a program simultaneously, each one of which can be driving at a given speed in a given direction.

Class Methods

While the attributes of a class give it the properties that we desire from the class, it is the *class methods* that make the **Car** class *perform* the way it should. In the case of the class, our class methods will carry out actions that we expect from a car such as turning, accelerating, braking and stopping. Let's create and define some of these class methods.

Double-click on the right side of the **Car** class icon. A window will open entitled **Car** but also containing a method icon in its title bar. Create a method icon and name the new method **Stop**. This method, which can be called only by objects created from the **Car** class, will be called when we want to slow down a moving **Car**.



Figure 10.7: The Stop class method of the Car class

Open the **Stop** class method's case window. Notice that the window's title shows both the class name (**Car**) and the name of the method called (**/Stop**), as seen in Figure 10.8.

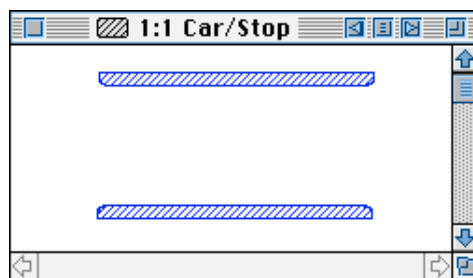
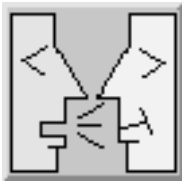


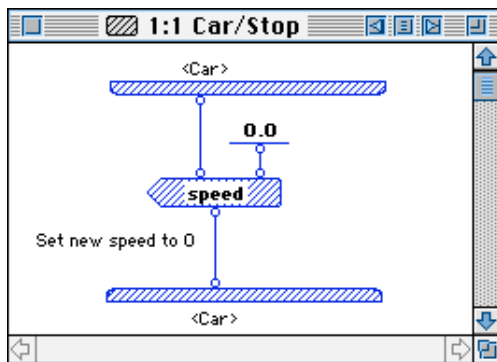
Figure 10.8: The case window of the Stop class method



By The Way...

The “/” between “Car” and “Stop” is not there simply to separate these two words. It has a symbolic meaning as well. Because class methods and Universal methods can have the same name, there must be a way to differentiate them when calling a method to be executed. When an operation has a leading “/” character, it signals Prograph that a *class method* is being called. The “/” is shown in the class method’s case window as a visual cue for us to remember the “/” when we call the class method. We’ll have more to say about this later.

Complete the **Stop** class method’s case window as shown in Figure 10.9. The C++ language version of the **Stop** method is shown in the figure as well. Note that in C++, we can specify data members as being “public” or “private”. Public data can be accessed directly, while the access of private data is restricted. While data access restriction is quite useful, the current version of Prograph CPX does not have such a mechanism -- all attributes are *public* and therefore directly accessible to all users of the class, as well as to all subclasses derived from the class. Perhaps future versions of Prograph will address what is one of the few shortcomings of Prograph. At present, however, you can at least hide some of the code of your classes (but not its data) by removing their source code before distributing them with the *Make Execute Only* option.



```
class Car {
public:
    void Stop( void );
private:
    float speed;
};

void Car::Stop( void )
{
    speed = 0.0;
};
```

Figure 10.9: The completed Stop class method with its C++ language equivalent

The **Stop** class method simply *sets* the value of the **speed** attribute to 0.0. How do we do this? Look at the items in the **Ops** Menu. One such item looks like this:

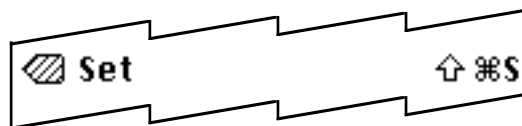
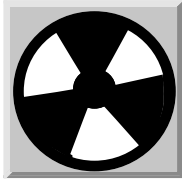


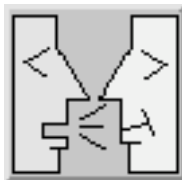
Figure 10.10: The Set operator menu item

This item converts a simple method into a *Set operator* which, as its name suggests, *sets* the value of the attribute that has the same name as this method. In the **Car** class' **Stop** method, we call a set operator labeled **speed** to give the **speed** attribute a value of 0.0.



Warning!

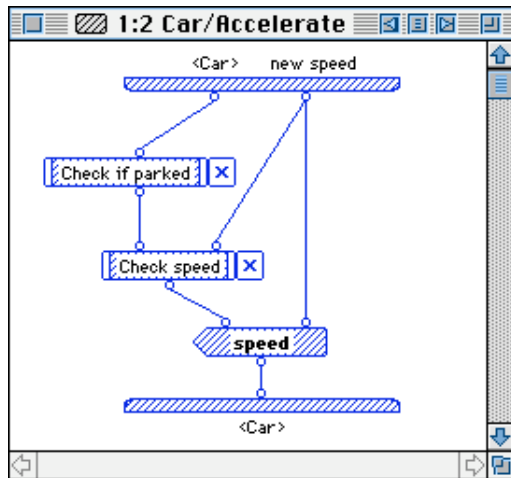
Prograph is *case-sensitive* just like C++. This means that naming the first get method of Figure 10.11 “Gear” instead of “gear” will produce an error, since “Gear” and “gear” will be interpreted as signifying different attributes.



By The Way...

Note that the input and output bars of the **Stop** method each have one node. The input node expects that an object of type **Car** is fed into the method. The output node returns the **Car** object from the method. Why do we need to do this? Actually, we don't need to do it. But writing class methods in this manner has one strong advantage. It allows us to “string together” many calls to the class methods of the **Car** class, feeding the **Car** object into the first class method call only. All subsequent class method calls will just “pass along” that same object. We'll show you how this works a little later in the chapter.

Create two more class methods -- **Brake** and **Accelerate**. Complete the **Accelerate** class method's first case as shown in Figure 10.11. This class method is called when you want to increase the speed of the **Car** as it drives, just as stepping on the gas pedal of a real car would do.



```
class Car {
public:
    void Stop( void );
    void Accelerate(
        float newSpeed );
private:
    float speed;
    char gear[ 10 ];
};

void Car::Accelerate(
    float newSpeed )
{
    if ( !CheckIfParked() )
        DoCase2();
    else if ( !CheckSpeed() )
        DoCase2();
    else speed = newSpeed;
};
```

Figure 10.11: First case of the Accelerate class method and its C++ language equivalent

The code of the **Accelerate** class method first enters a local method named **Check if parked** (see Figure 10.12) which, as its name suggests, checks whether or the **Car** is in the “*park*” gear by reading the value of the **gear** attribute. After all, you can’t speed up a car when it is not in “forward” or “*drive*” gear. If you try to do so, a warning message is presented.

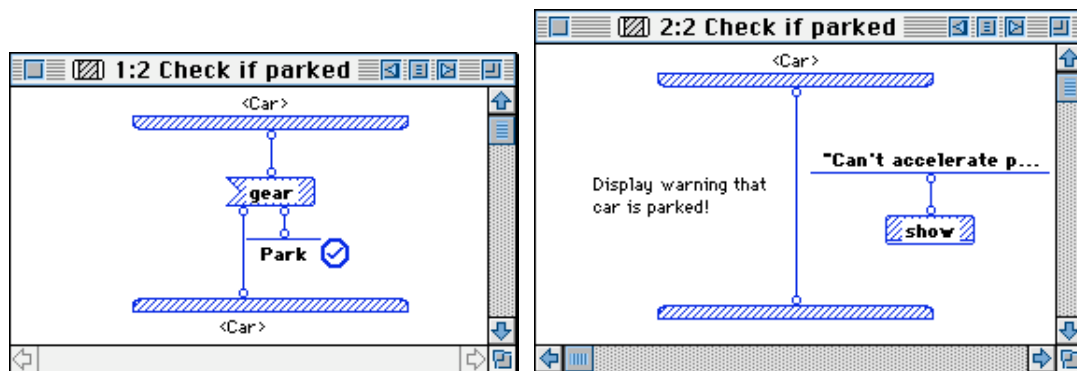


Figure 10.12: The Check if parked local method

The value of the **gear** attribute is read with a *Get* operator. *Get operators* are the opposite of *Set operators* -- they retrieve the current value of an attribute of a class. They are created from simple operators by means of the **Get** item in the **Ops Menu**, shown in Figure 10.12.

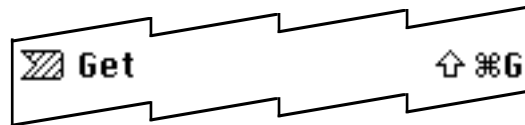


Figure 10.13: The Get operator menu item

What's that strange control on the match test in the Check if parked local method? This is a special control called a "*Fail*" control (also shown in Figure 10.14). Let us explain why this particular control is needed. What if we had placed a regular "next case", "terminate" or "finish" control on this match? When the match succeeded (that is, when the value of **gear** was equal to "Park"), the second case of the Check if parked local method would execute and an error message would be presented. Fine -- that's what we want. But what happens next? The **Accelerate** method will continue to execute its second local method, then reset the value of **speed** just as if nothing had happened within the Check if parked local method. The **Accelerate** method is unaware of the results of the match test within its local method.



Figure 10.14: A Fail control on a match test

Clearly, we need a way to tell the **Accelerate** method that the Check if parked local method's match test succeeded. This is where the "*Fail*" control comes in. It not only makes the local method change its flow according to its outcome, but it also *signals the method that calls the local method* about the result of its match test. If the match test is set off (in this case, if it *succeeds* since the test has a check mark in it), the "*Fail*" control is triggered. In this manner, the result of the match is *propagated* from the local method to the method that called that local method.

By attaching a "next case" control to the local method's operator icon in the **Accelerate** first case window (see Figure 10.11), we can make the **Accelerate** method change its course of action depending upon the outcome of the match within the Check if parked local method. If the match test succeeds (that is, the gear is indeed set to "Park"), the second case of the Check if parked local method will execute to display a warning message. The success of the test will also trigger the "*Fail*" control. Next, the "next case" control on the Check if parked local's operator icon will sense that the "*Fail*" control was set off. It will then make the method enter its own second case (shown in Figure 10.15), which will skip the Check speed local and the **speed Set** operator. The final outcome is that the success of the match test within the Check if parked local method will cause the **speed** attribute to be left unchanged.

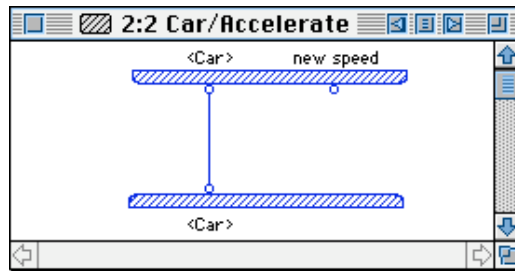


Figure 10.15: Second case of the Accelerate class method

After this, the **Accelerate** method determines if the requested new speed really exceeds the current speed of the **Car**, using a similar *Get* operator labeled **speed** to read the current value of the **speed** attribute (Figure 10.16). If this test succeeds, the **Accelerate** method will finally set the value of **speed** to the new speed using a *Set* operator.

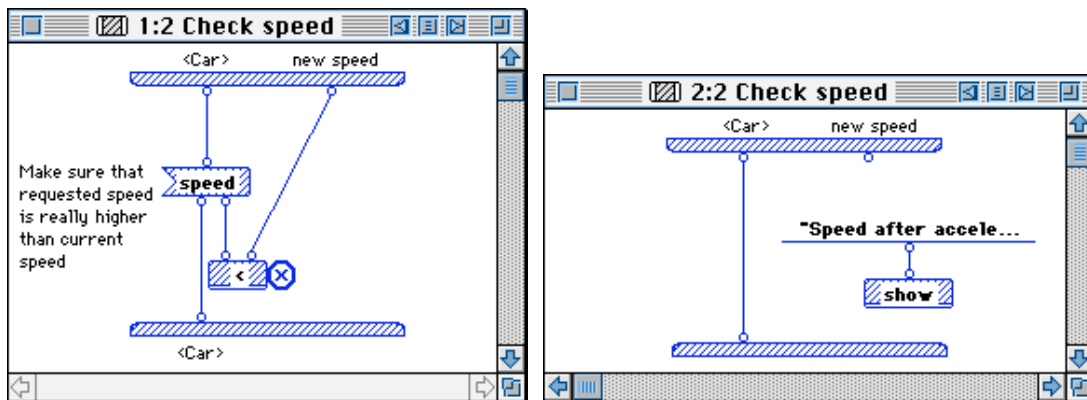


Figure 10.16: The Check speed local method

Proceed now to the **Brake** class method (Figure 10.17). This method is very similar to the **Accelerate** class method, except that the speed of the **Car** is decreased rather than increased. First, the requested new speed is compared to the current value of the **speed** attribute. If it is truly lower than the current speed, the value of the **speed** attribute is reset to its new value.

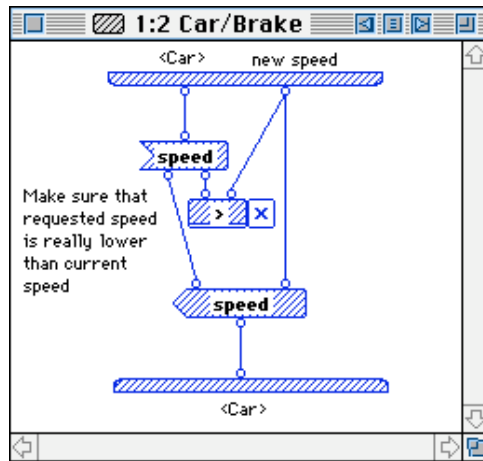


Figure 10.17: First case of the Brake class method

If the requested speed is in fact higher than the current speed, a second case is entered (Figure 10.18) in which a warning message is presented to the user.

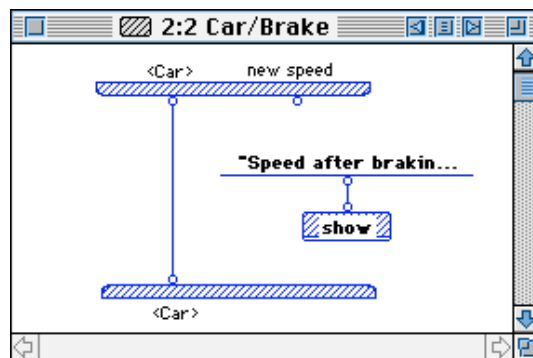


Figure 10.18: Second case of the Brake class method

User-defined Get and Set Methods

In each of the above class methods, we simply set the value of the **speed** attribute with a new value. This is a pretty straightforward process. But sometimes we cannot just set an attribute to a new value. An example of this is the **direction** attribute of the **Car** class. The direction of the **Car** is encoded as an integer number from 0 to 359, as in the number of degrees in a circle. A value of 0 represents *North*, 90 represents *East*, 180 is *South* and 270 is *West*. This representation facilitates the process of *turning* the **Car**, since turning to the *right* would simply involve *adding* 90 to the current value of **Car**, and turning to the *left* would mean *subtracting* 90. Unfortunately, this simple way to turn the **Car** presents a problem. If the **Car** is facing *West* (direction is 270) and we turn to the *right*, the value of **direction** will now be 360 instead of the expected value of 0 which represents *North*. Similarly, if the **Car** is now facing *North* (stored as a **direction** value of 0) and we turn to the *left*, the value of **direction** will now be -90 instead of the expected value of 270 that represents *West*.

Clearly, we need a way of setting the **direction** attribute that will correct for this problem. Prograph gives us this way -- it allows us to *redefine* the process of setting or getting the value of an attribute with custom *Get methods* and *Set methods* of your design that will be executed instead of the built-in Get and Set operations. Why would we want to do this? We use custom Get and Set methods to *automatically* carry out some action *every time* we set or get an attribute's value. These automatic actions might prevent possible errors during program execution. For example, what if an attribute of a class holds an integer. Before we can set a new value for the attribute, we might check that the new value is really an integer. This could be done by explicitly adding code that checks if the data is an integer *before every call to the attribute's Set operation*, but then the programmer would have to remember to do this every time or face the consequences. Better yet, you could simply add this check to a *custom Set method*. The custom Set method would then perform the check and then assign the integer value to the attribute *automatically*. Because the check is *within* the new Set method that will be called each time we want to change the attribute's value, we can never accidentally forget to perform the check.

Let's write our own *Set method* for the **direction** attribute. In the process of writing this method, we'll show you a way to write more complicated "if-then" control constructs.

Click in the class method window and create a new class method. Call this method **direction**, the name of the attribute we want to set with the method, and convert the method to a *Set method* by selecting the Set item from the Oper's Menu. Notice that the icon for the method changes to one that resembles that of the built-in *Set* operation, as shown in Figure 10.19.



Figure 10.19: Icon for a custom Set method

Fill in the method's case window's code diagram as depicted in Figure 10.20.

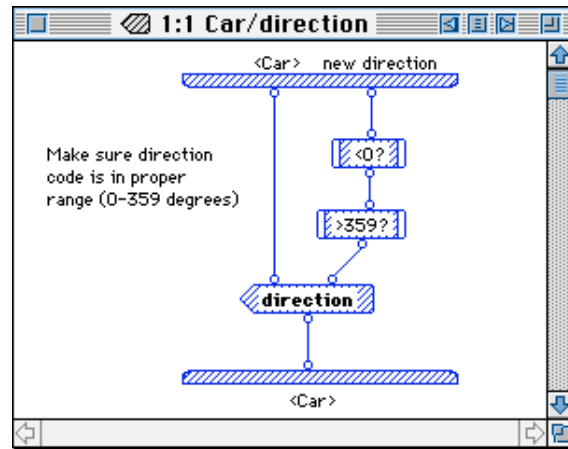


Figure 10.20: The direction Set method

In C++, you would have to write a function in the `Car` class called `SetDirection()`, always remembering to call this function when writing a new value to the `direction` data member. If you forget to call this function, `direction` might be incorrectly set. In Prograph, you *don't* have to make a method with a new name -- Prograph just calls your custom *Set* method instead of the built-in *Set* operation with the same name.

The custom `direction` method accepts a `Car` instance and a user-supplied direction and, after a few tests on its value, ultimately stuffs this new value into the `direction` attribute. You might have noticed that there's a call to *Set* the `direction` attribute *inside our own direction Set method*. What's happening here?

Remember that we are writing a method to replace the default *Set* operation for the `direction` attribute. At some point in our new method's code, we'll still have to perform the actual act of setting the value of the `direction` attribute. That is, we still need to use the *built-in direction Set* operation (the one that was created automatically for us when we defined a `Car` class containing the `direction` attribute). Unfortunately, *our direction* class method and the *built-in direction* operation have the *same name* -- there must be a way for Prograph to *tell them apart*. We do this solely *by the way we call Set* in our program.

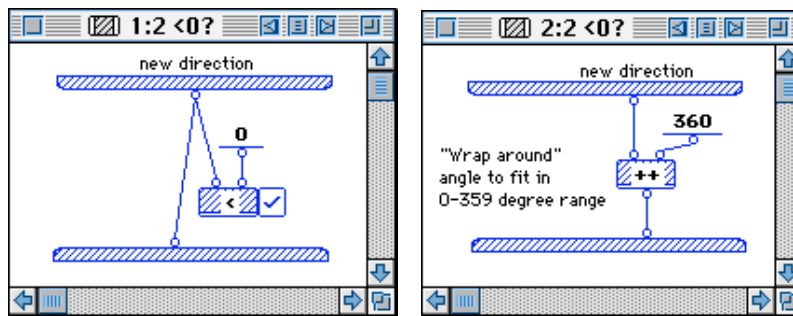
If Prograph encounters a *Set* operation whose attribute name is preceded with a leading "/" (in the case of our attribute, `/direction`), *your new custom Set method* will be executed. If, instead, the *Set* operation *lacks* the leading "/" (that is, `direction`), Prograph will default to executing the *built-in* operation and *not* the special method we wrote. The table shown in Figure 10.21 summarizes this notation.

<u>Custom Get method</u>	<u>Call built-in Get operation</u>	<u>Call custom Get method</u>
absent	methodName	
present	methodName	/methodName

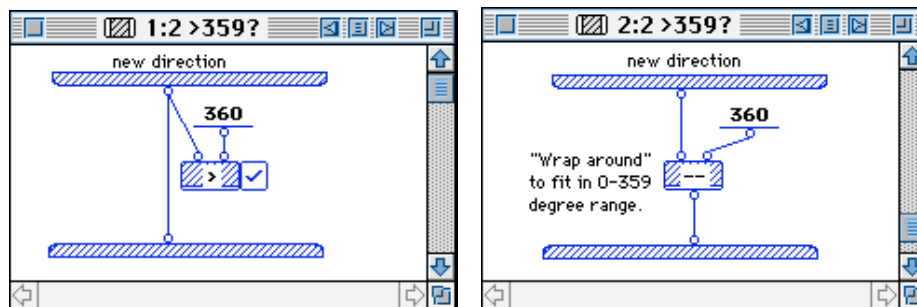
Figure 10.21: Calling a custom Get method versus a built-in Get operation

So in our new *direction* method, we must call the default *Set* operation to do the actual setting of the *direction* attribute. Everything else in the window is just all the *extra* stuff we want done automatically whenever we choose to change the *Car's* direction. This extra code includes two *match* tests within local methods. These tests will ensure that the value of the new direction is within the proper range -- from 0 (due North) to 359. If the new direction value input into the method is less than 0 or greater than 359, the local methods will execute additional cases to correct it by "wrapping around" the value of the direction. That is, if the value is greater than 359, it will be "wrapped around" to 0; if it's less than 0, it's "wrapped around" to 359.

The first local method, whose code is shown in Figure 10.22, checks to see if the desired direction is less than zero. If the match (<) succeeds, a second case must be entered in which 360 is added to the directional value.

**Figure 10.22: The <0 local method**

The second local method is depicted in Figure 10.23. It checks to see if the desired direction is greater than 359. If the match (>) succeeds, a second case is entered to subtract 360 from the directional value.

**Figure 10.23: The >359 local method**

Let's create the *Turn* class method now. This method will be used to steer the *Car* to the left or to the right. The special precautions we took in "wrapping around"

inappropriate values of the **direction** attribute in the **direction** class method will come into play now to simplify the **Turn** class method.

When directions are stored as numbers from 0 to 359, turning the **Car** to a new direction becomes simple. To turn the **Car** to the left, we subtract 90 from the direction (see Figure 10.24). This is why the **direction** class method added 360 to the **direction** when we tried to set it to anything less than zero, such as -90.

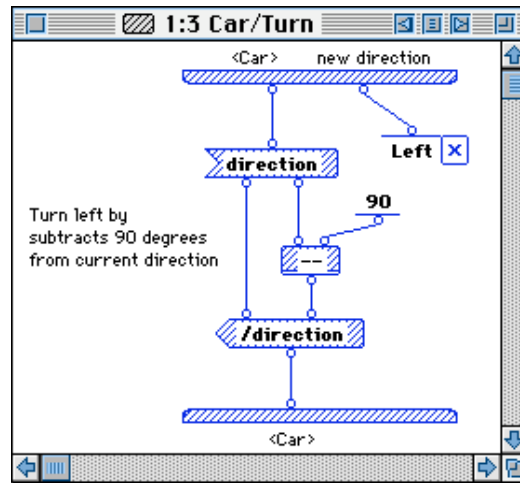


Figure 10.24: First case of the Turn method

If the turn is to the right instead, the current direction code is incremented by 90 and the new value is put back into the **direction** attribute, as seen in Figure 10.25.

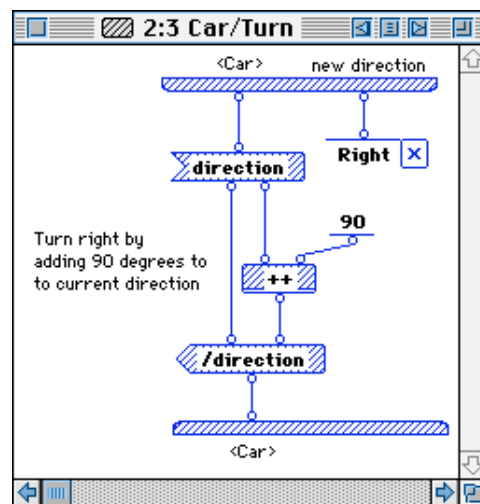


Figure 10.25: Second case of the Turn method

If the user attempts to turn in any other direction, a third case is entered that displays a warning message (Figure 10.26).

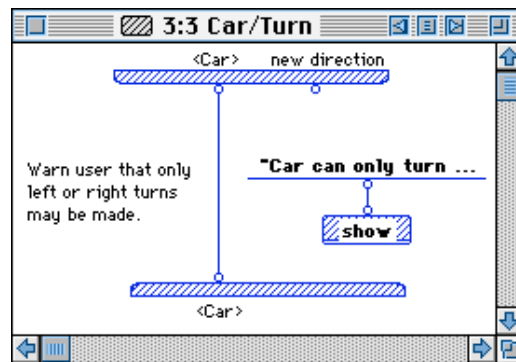


Figure 10.26: Third case of the Turn method

The Shift Gears class method changes the current gear of the Car by setting its gear attribute (see Figure 10.27).

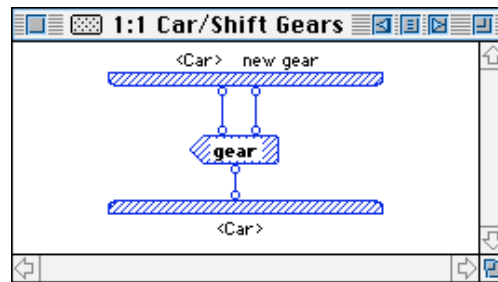


Figure 10.27: The Shift Gears method

Setting Default Attribute Values

We've now completed all of the Car class methods, but we have one last step to complete. When we create an object from a class, what are the precise values of that object's attributes? What if we want a new object to have *specific values* for its attributes?

We may set *default values* for each attribute by editing them in the Attributes window. In Figure 10.28, we have reset the attributes of the Car class to the values we want them to have by default. Now, every time we create an object from the Car class, it will have all of its attribute values set to these default settings *automatically*. We'll never have to worry about accidentally starting to use the object with a random or unknown attribute value. In procedural code, it's easy to forget to call methods that do initialization of data. This may result in the use of inappropriate data values that can cause the program to run incorrectly or crash. Default attribute values are just one more advantage of using object-oriented programming. Objects created from a class are *automatically* initialized without the need for you to explicitly call such an initialization method yourself for each and every object you create.

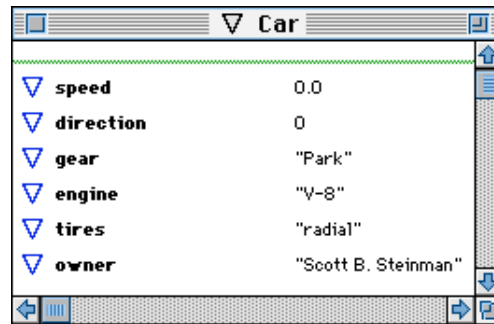


Figure 10.28: Setting default values for the attributes of the Car class

Putting Objects to Use in Prograph Programs

Our **Car** class, although fairly simple, is complete and ready to be “plugged into” a program. How do we use the object? It's fairly simple. We'll write a Test Car universal method that will create an object from the **Car** class, then get the **Car** to “drive around”.

Open the Universal window if it's not already open. Create a universal method called Test Car. Open the code window and click in it to create a new program element. Now, while the operation's icon is highlighted, select the Instance item from the Opers Menu. The Instance menu item converts an untyped “plain” Prograph operation into an *instance generator* that will create an *instance* of a class as its output -- in other word, an *object*. This is shown in Figure 10.29.

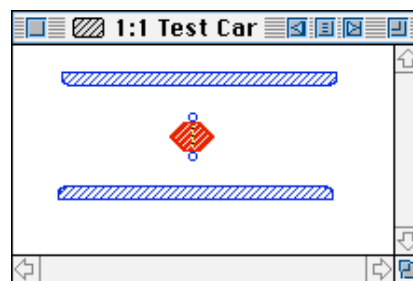
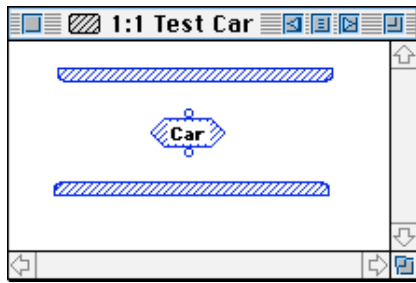


Figure 10.29: Creating an instance of a class

Name this new instance (or *object*) **Car** to let Prograph know that this object will be derived from the **Car** class (Figure 10.30). The instance icon denotes more than just the creation or *instantiation* of an object. After instantiation, Prograph can also automatically call an *instance or initialization method* if you write one. Although you can set attribute values simply by defining default values for attributes, instance methods allow you to do more than this, such as allocate memory automatically when the object is created. In this respect, instance methods are like C++ class constructors. We'll show you one powerful way that an instance method can be applied in the next chapter when we create practical classes that you can reuse in your programs.

Note that the **Car** instance generator icon has one root and one terminal node. The root node of the instance generator returns the new object we've just created. The terminal node is used optionally for feeding *input parameters* into the instance generator. These parameters, sent into the instance generator in list form, are values to be placed into *each of the newly-created instance's attributes*. This is simply another way to set the attribute values at run-time without having to write an initialization method to set each value individually. This method of initialization can be quite useful -- we'll show you a powerful way to take advantage of it in Chapter 13.



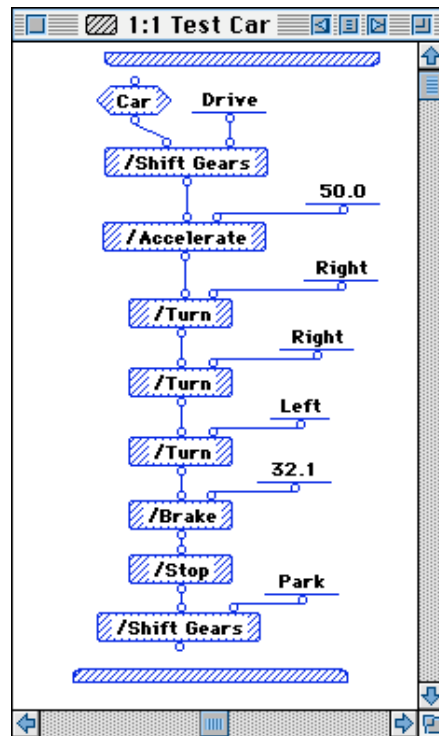
```
void TestCar( void )
{
    Car myCar = new Car;
}

// Constructor

Car::Car( void )
{
    // Set the Car's data values
}
```

Figure 10.30: Generating an instance of the Car class, with its C++ language equivalent

After creating a **Car** object, we can get the new **Car** to perform the actions we desire of it by sending it *requests to perform actions*. These “requests” are in fact *calls to that object's class methods*. Complete the Test Car method code diagram as shown in Figure 10.31. Now you can see why we wrote each class method to accept a **Car** object as an input, then output the same object again when finished. This allows us to send one method call after another to the same **Car** object. In other words, we can pass one **Car** instance directly from one class method call to the next. Our code diagram is simpler and easier to understand. It can be seen easily that the Test Car method is taking one **Car** and asking that **Car** to do several actions in sequence.



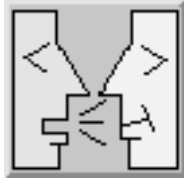
```

void TestCar( void )
{
    Car myCar = new Car;

    myCar.ShiftGears( "Drive" );
    myCar.Accelerate( 50.0 );
    myCar.Turn( "Right" );
    myCar.Turn( "Right" );
    myCar.Turn( "Left" );
    myCar.Brake( 32.1 );
    myCar.Stop();
    myCar.ShiftGears( "Park" );
}

```

Figure 10.31: The Test Car method for creation and use of Car objects and its C++ equivalent



By The Way...

We've only used *one* particular way of calling class methods in the example above. The `/method` syntax is a *data-determined* method reference, in which the method to be executed is found within the object that arrives on the leftmost root node of the called method. But there are also other ways to call class methods. If one class method is called by another class method using the syntax `//method`, the method call is *context-determined*. The called method is within *the same class* as the method calling it. Finally, a class method may be called as `class/method` - an *explicit* class method call - where we explicitly state which class contains the called method.

The simplicity of the Test Car method shows the beauty of using OOP in Prograph programs. Once the **Car** class has been completed, we just create an object from it and send the object requests to do things like speed up or turn. To *reuse* the completed **Car** class in another program written by yourself or somebody else, never needs to know ever again *how* each action of the **Car** is carried out. All they have to do is send requests to the **Car** to perform these actions. The internal workings of the **Car** have been *encapsulated* or *hidden from the user*. In this example, the code of the Test Car method does not need to access the inner workings of the **Car** in order to make the **Car** do its stuff. Object-oriented programming has made it easy to reuse the **Car** over and over again in new programs.

Exercise 10.1:

Design and create a class called **Animal** that has the major characteristics of animals such as a tail, color, a number of legs, size, etc. Provide methods to allow the class to act like an animal. Test the class with a universal method that creates an object of type **Animal**, then has it perform the actions expected of an animal. Use the show primitive to display what the animal is doing.

Remember that although all objects created from a given class share the common properties and actions of the class, each object does not have to have the same values for each of its attributes. Just to demonstrate, try out the Test 2 Cars method shown in Figure 10.32. This method creates two independent **Car** objects, and sets the **speed** and **direction** attributes of each to different values by accelerating to different speeds and turning in different directions. So even though each of these two objects is derived from the same **Car** class, the objects do not have to be identical in *every respect* to still be cars. If their attributes were always identical, you couldn't have two models of car or drive the two cars differently.

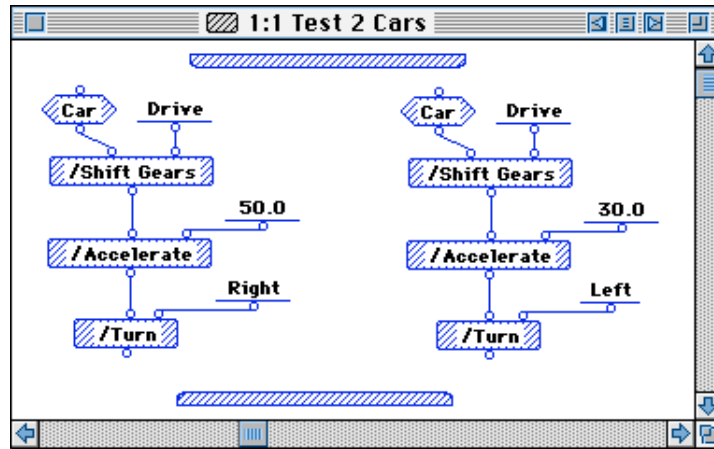


Figure 10.32: The Test 2 Cars method for creation and use of two Car objects

Summary

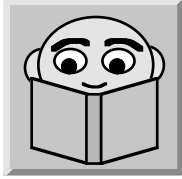
Classes are the backbone of object-oriented programming. In this chapter, we have shown you how to create classes in Prograph, by defining both their attributes and class methods.

- When attributes are defined, they may be given *default values*. Every object created from the class will have its attributes automatically set to the default value.
- Special *Get and Set operations* are automatically provided by Prograph to read the current value of an attribute or write a new value to it.
- We may *replace* the Get and Set operations for an attribute with our own *custom Get and Set methods* that will perform extra actions of our choice every time we read an attribute's value or try to give it a new value.
- *Instance generators* exist solely for the creation of instances (objects) of classes. We can have the instance generator automatically call an *instance or initialization method* of our own design to allocate memory or perform other special tasks. In the next chapter, we'll examine ways to take advantage of these methods for our own use.

We also introduced a new control for logical match tests in this chapter:

- The *fail* logical control not only tests for the success or failure of a logical match test, but it can also transmit the result of the test to higher-level methods to terminate execution of code dependent upon the outcome of the logical test.

So far, we've used only a single simple class in isolation. We now turn away from the use of a single class to the creation of more complicated classes using *inheritance* and *composition*.



**For More
Information...**

Unfortunately, it is beyond the scope of this book to show you how to write huge programs using dozens of objects that send requests for action back and forth between themselves, although some details of multiobject programs will be discussed when we turn to the Application Builder Classes later on. Large programs such as these are typically undertaken by *programming teams* at software companies. All we can do is get you started with the techniques used most often in smaller programs written by a single programmer. For guidelines for designing large-scale OOP programs, see the book “Designing Object-oriented Software” by Rebecca Wirfs-Brock, Brian Wilkerson and Lauren Wiener (Prentice-Hall, 1990).