

9. Principles of Object-Oriented Programming

Overview

The principles of top-down *procedural programming* we reviewed in the previous section of this book will now be extended and modified as we turn to a new style of programming -- *object-oriented programming* (abbreviated as “OOP”). Object-oriented programming, like procedural programming, is primarily a strategy for breaking down large, complicated programming tasks into a set of smaller, simpler, more manageable components. The difference between these two programming approaches is the *particular* way in which we break down the programming task. The object-oriented paradigm brings with it several major benefits, not the least of which is the ability to take advantage of Prograph’s *Application Builder Classes* to easily design and manage user interfaces (windows, menus and dialogs). This chapter will focus on what OOP is, why you should use it, and when it should be used.

Why Use Object-Oriented Programming?

Why can’t you just use procedural programming? Why use OOP at all? Well, you *can* still use procedural programming -- some programmers would argue that certain types of programs may be better suited to procedural programming, such as math-intensive programs. In fact, many large commercial applications are still programmed in a procedural style. However, there are problems with continuing to write large programs in a procedural style.

Most of the time spent programming isn't really devoted to writing programs -- it's spent *modifying* programs to fix errors and to *change what programs do* to fit new needs. As you build larger and larger procedural programs, the programs get much harder to manage and modify to handle new chores or add new features. Think about how many of the applications you own which are not updated regularly, or when they are finally updated are still very much full of bugs. This problem is a symptom of what has been called the “*software crisis*” -- the need to spend more and more time merely maintaining and fixing applications, let alone improving them. Why has the software crisis occurred? Simple -- the more complicated the program, the less efficient the procedural programming style becomes in terms of programming effort.

Procedural program design concentrates on breaking apart a large task into several, sometimes hundreds or thousands, of methods. Unfortunately, these numerous methods are completely independent of each other, as suggested by Figure 9.1 -- essentially *unaware* of what other methods do and *upon what data* the other methods perform their actions. The burden is placed on the *programmer* to ensure that methods are called in the correct order, and that each method uses the proper data input. If not, the program will provide incorrect answers or, even worse, crash. The more methods you add to the program, the harder it is to manage potentially harmful interactions between methods in different parts of the program. Fixing a bug or adding code to one part of the

program might require changing methods in other parts of the program too. Now imagine what happens when a bug occurs! How do you find it in this mass of independent code?

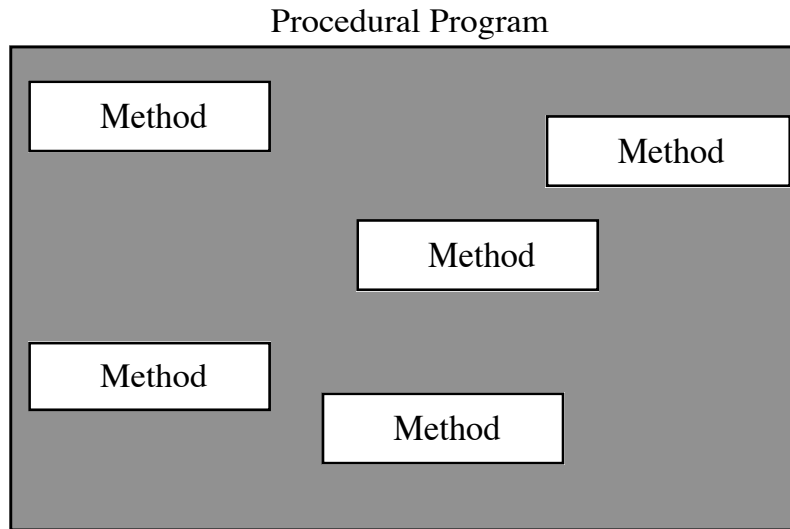


Figure 9.1: Organization of a procedural program

It's here that OOP can help. Object-oriented programming is a different way to break down a program into small, manageable modules that accomplishes many of the goals of well-written software. First, the modules in OOP programs wind up being more *adaptable* and *reusable*, so we can then *pick and choose* them to build new programs, then *extend* them to suit new situations. Second, the modules are also better "self-contained", which improves their *reliability* by reducing the possibility of errors within them propagating throughout the program that contains them.

With OOP, programs are not just collections of unrelated methods, but are systems of interacting *self-contained modules of data and methods* called *objects* which can "ask" each other to perform actions (see Figure 9.2). The objects are defined by *classes* -- descriptions of the data and methods that each object contains. Program design involves for the most part planning what we want the objects in our program to be able to do (by themselves or in conjunction with other objects). We do this by creating definitions called *classes* that describe these objects. So while procedural programming concentrates only on which *operations* should be done and in what order, object-oriented programming focuses on which *data modules (objects)* should be used in a program, and how each data module will act *on its own and in cooperation with other data modules*.

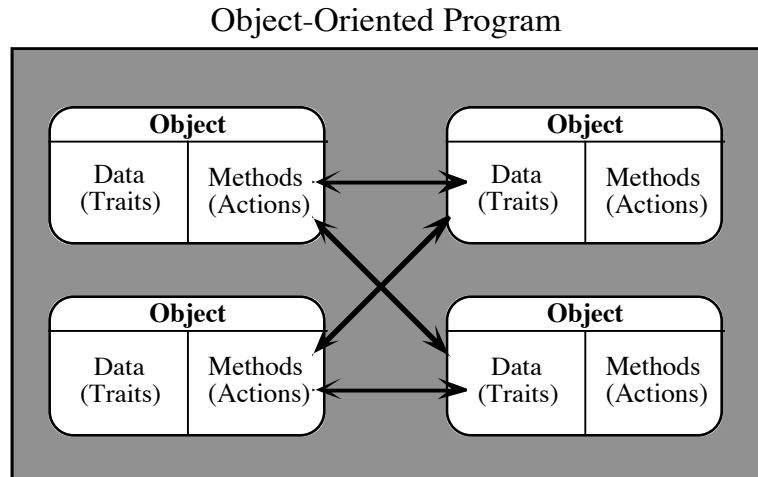


Figure 9.2: Organization of an object-oriented program

As you can tell from the above discussion, object-oriented programming encourages us to spend more time on *program design* and less on writing the *precise* operations needed to perform the tasks of our program. While this may make the design phase of programming last longer, the benefits of this extra forethought and planning far outweigh its costs. Object-oriented design saves us a great deal of time later when we must debug or modify the program. For example, if you have a class called **File** in your program that handles reading and writing files, and a file access bug exists, you know *exactly* where the problem must be.

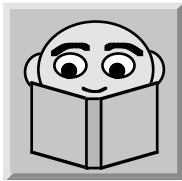
The compactness and modularity of objects and classes also has two major benefits. First, it makes objects and classes more *reusable* than are independent methods - - it is easy to “plug in” an existing class into a new program and have the program work correctly. Classes may therefore be used over and over again to create objects in new programs. Second, it is also easier to *change* the inner workings of a class -- how it is implemented -- without disastrous effects on the rest of a program. This makes it easier to experiment with writing more efficient code for an object without having to totally rewrite the rest of the program to account for the changes in the class. Classes allow extensive *prototyping* of programs.

The OOP model of software development fits in very nicely with user interface design. User interfaces are based upon several OOP concepts. The user interface is composed of interrelated menu, window, dialog, alert and control objects interacting and communicating with one another to present information to the user in a friendly way.

All of this unbridled enthusiasm about object-oriented programming does not imply that OOP is a magical solution to all programming problems -- no programming paradigm is. You can make errors in writing OOP programs just as you can in other programming styles. However, the programming style of OOP programs helps avoid several problems and helps promote code reuse. That in itself offers more than enough benefits for the programmer.

Object-oriented programming is considered by many to be the programming style of the 90's. Microsoft has embraced OOP for its Windows NT operating system. NeXT has bet its entire future on an object-oriented operating system for UNIX and Intel-based computers. Apple Computer has increasingly warned software developers that they must employ OOP if they are to take full advantage of the new Power Macs. Apple and IBM's forthcoming Taligent operating system for Power PC and Power Mac computers has been written from the ground up as a collection of objects.

Luckily, Prograph not only supports object-oriented programming, it encourages its use by making it simpler. Prograph not only allows you to create and use objects with visual design tools, but it also provides a rich object-oriented library called the *Application Builder Classes* for creating and managing user interfaces. We'll discuss the Application Builder Classes in Chapters 14-17.



**For More
Information...**

For an in-depth discussion of the concepts of object-oriented programming, we recommend that you read "An Introduction to Object-Oriented Programming" by Timothy Budd (Addison-Wesley Publishing Co., 1991) or, for Macintosh-specific OOP programming, "Object-Oriented Programming for the Macintosh" by Kurt Schmucker (Hayden Books. 1986).

Solving Problems by Using Classes and Objects

Object-oriented programming is a more natural way than procedural programming to *represent real-world concepts*. The key building blocks of object-oriented programming are *classes* and *objects*. *Classes* define what a *set* of real-world *objects* have in common, using both data and methods.

A class can be thought of as a "*template*" for the creation of individual "generic" objects. A class describes what *all objects of that type* have in common -- what makes the object that particular type of object. *Objects* are the *particular instances* of a given class. Each object possesses the common set of *attributes* defined in the class, but may differ in the *exact values* given to each attribute. One way of thinking about this is that a class is like a *cookie cutter* that describes the shape of the cookies you'll create. The objects are the individual *cookies* themselves. They must have the basic properties of cookies and must have the shape defined by the cookie cutter, but they don't all have to be *precisely* alike in *all* respects -- one may be chocolate, another vanilla, another lemon and one may be eaten or cooked and another not.

Let's illustrate classes with a concrete example. Let's say that you need to write a car racing game program that simulates driving different models of automobiles. To adequately make a computer program behave like a car, we create a *class* that represents the common characteristics of *all* cars. First, all cars perform similar *actions*. For

example, cars *can* steer, move forward or in reverse, accelerate, slow down, or stop, but they *can't* do things like fly or walk. These actions form the *methods* of the class.

Second, all cars are built from a common set of *traits* -- for example, the parts of the car such as its engine, tires, transmission, and so on -- that remain constant for a given car and distinguish it from other cars. Third, each individual car has its own unique *state* - the current speed at which it is traveling, its current owner, etc. -- that can change for a given car. These common traits and state contribute to the *data* part of the class.

It is the *common characteristics and actions* of cars that are defined in a *class* that could represent any type of car. We could create a class called **Car** that contains both code components called *class methods* (the *actions* that the car is able to do), and data components called *attributes* that define the *properties* of the car -- both its *traits* (what makes the car uniquely a particular kind of *car* and not a cat or dog) and the *state* of the car (the condition of the car at this moment). Figure 9.3 shows the components that make up a class.

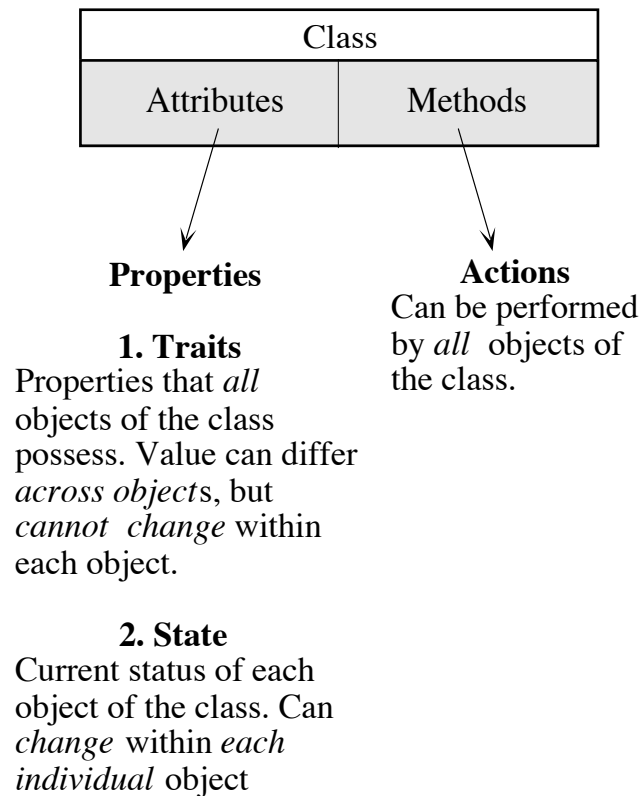


Figure 9.3: The components of a class

Returning to our car example, while the **Car** class just defines what *all cars have in common*, *objects* created from the **Car** class are the *individual distinct cars themselves*, such as *one* particular person's Ford or *one* given Ferrari (see Figure 9.4). That is, while the **Car** class is a *general description or model* for what *any* car is, objects created from

the **Car** class are *individual* unique cars differentiated by what makes *one* car different from another. One given object, such as My Ford or My Ferrari, is still a type of car, exhibiting the common features and actions of the class **Car**. For example, each object has an engine and can drive. However, each of these two objects is *not exactly like the other*. The precise *type* of engine in each will differ, and each may belong to a *different* owner, even though each is still a car. So while our My Ford and My Ferrari objects both have the common characteristics of **Car** -- they possess all of the *attributes* of a car such as engine, tires and owner -- the *particular value* of the **Engine**, **Tires** and **Owner** attributes may differ in these individual objects.

The *class methods* that each object can access, that is, the *actions* each can do, will be the *same* for each object of this class, since *all* types of car should do the same actions -- accelerate, brake or turn. If the My Ford or My Ferrari objects *could* perform totally different actions, then they would really represent different things. They would be objects of two *different* classes.

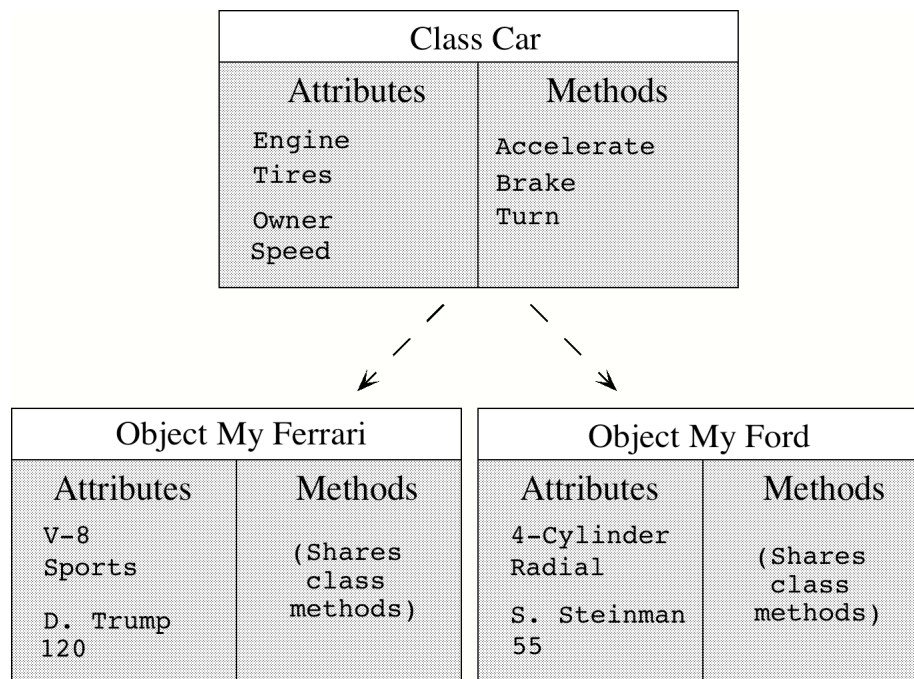


Figure 9.4: A class defines a set of objects

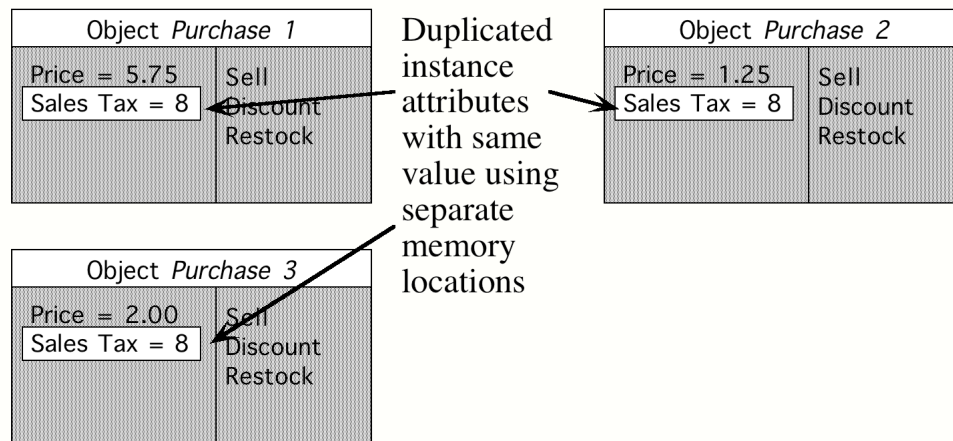
Exercise 9.1:

Design a **Housecat** class that would represent any house cat, using data attributes to store the traits and state of the cat and class methods to describe the actions that a cat can do.

The attributes of a class can be of *two* different types. In Prograph, these are called *instance* attributes and *class* attributes. *Instance attributes* are data elements of a class whose value may *differ in each object* created from this class; in other words, the

type of data we've discussed so far. *Class attributes* are special data elements whose value is the *same* in *every* object created from the class (see Figure 9.5). This value is stored only *once* in memory and is *shared by every object of this class*. Let's say we wanted to store a *sales tax rate* in a **Merchandise** class that would be the *same* for all **Purchase** objects created from the **Merchandise** class. It would be foolish to waste memory by storing the sales tax rate in memory within *every single* object of type **Merchandise** over and over again. By storing **Sales Tax** in a single memory location for all objects of type **Merchandise** as a single *shared* class attribute, we save memory storage.

Using Instance Attributes Only



Using Class Attributes To Share Memory

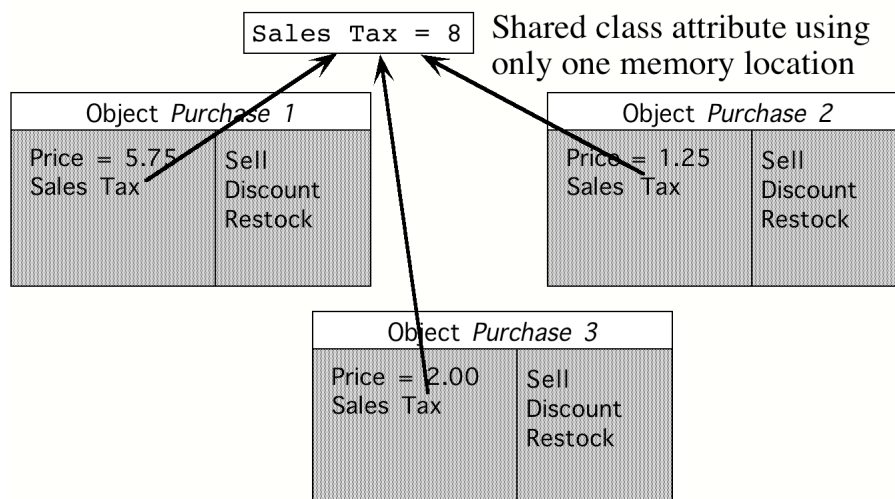


Figure 9.5: Instance attributes versus Class attributes

The packaging of code and data into classes and objects not only makes our programs easier to read, but has the added benefit of letting us locate program errors more easily. If a program using the **Car** class doesn't function properly because our

aFerrari and aFord objects are acting strangely, we know that the bug has to be within the data or code of the **Car** class, and not in some other part of the program. In procedural programming, on the other hand, an error in one method of the program can propagate errors to *other* parts. Finding *where* in the program the initial bug is located can often be very difficult.

The Class Interface

Classes and objects provide *data abstraction* -- the creation of *new data types* that represent real-world entities. You've already been taking advantage of user-defined data types in Prograph. The built-in number, text and list data types were designed by programmers at Prograph International. Numbers, text and lists do not automatically exist within the binary code of the computer -- they had to be modeled by programmers so that you could take advantage of them in your programs.

Classes let you define new data types that mimic real-world things like paint brushes or calculators, or concepts like data files and complex numbers. So long as you can accurately describe what something does, you can write a class to model it. Once you've written a class, the objects created from it can now be used as if it were a *built-in data type* like list or number. *Data encapsulation* is the term for the ability to use classes as new data types in programs without needing to know their inner workings. This is similar to the use of many real-life objects. For example, we can *use* a refrigerator without knowing how the freon in its coolant coils is compressed or how heat is exchanged from the box inside it to the coils. Similarly, it doesn't matter to a program *how* a user-defined data type like a list works, only that the list *behaves* the way it should.

The actions that a program may perform with objects of a given class forms what is called the *class interface* -- in other words, the class methods which may be called and the inputs required by these class methods. If programs only access an object through its class interface, the object will only do what it's supposed to do, and its data will be protected better from accidental changes.

In the example shown in Figure 9.6, an object is created from a **Counter** class. We may get an object of type **Counter** to initialize the value of its **countValue** attribute to 0 because the **InitializeZero** class method exists in the class interface. However, we cannot have the object's **countValue** attribute initialized to a value of 5 since there is no **InitializeFive** method. We have restricted the possible actions of objects of the **Counter** class to initializing **countValue** to 0, incrementing **countValue** and decrementing **countValue**, since these are the only methods in the class interface.

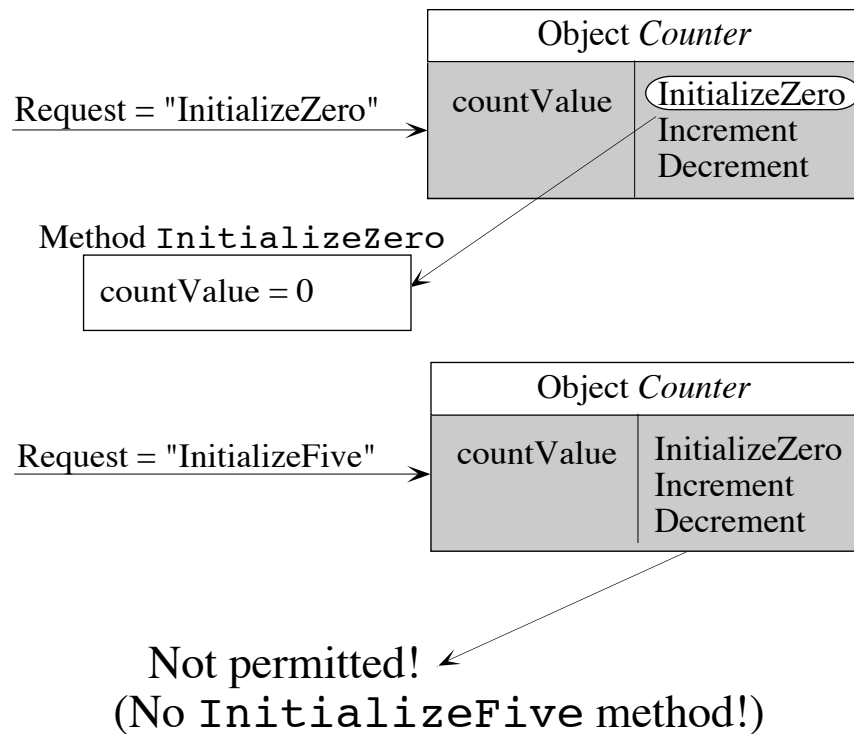


Figure 9.6: Enforcing the behavior of a class via its class interface

Communication With and Between Objects -- Requests to Perform Actions

Objects *intercommunicate* by “requesting” that specific actions be done (Figure 9.7). A program can therefore be thought of as a series of requests sent to objects or from one object to another. The objects are “asked” to perform actions until the program’s goals are reached. The object receiving the request has the *responsibility* to carry out the requested action, and it does so by executing one of its own *class methods*. In other words, an object-oriented program is a series of commands to objects (or from one object to another) to execute their own class methods.

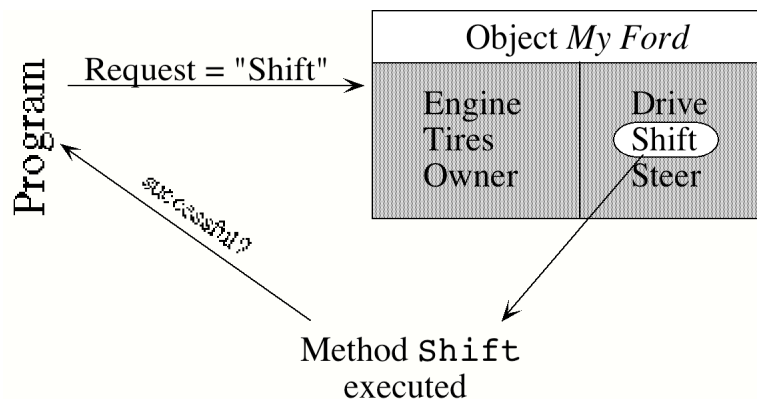
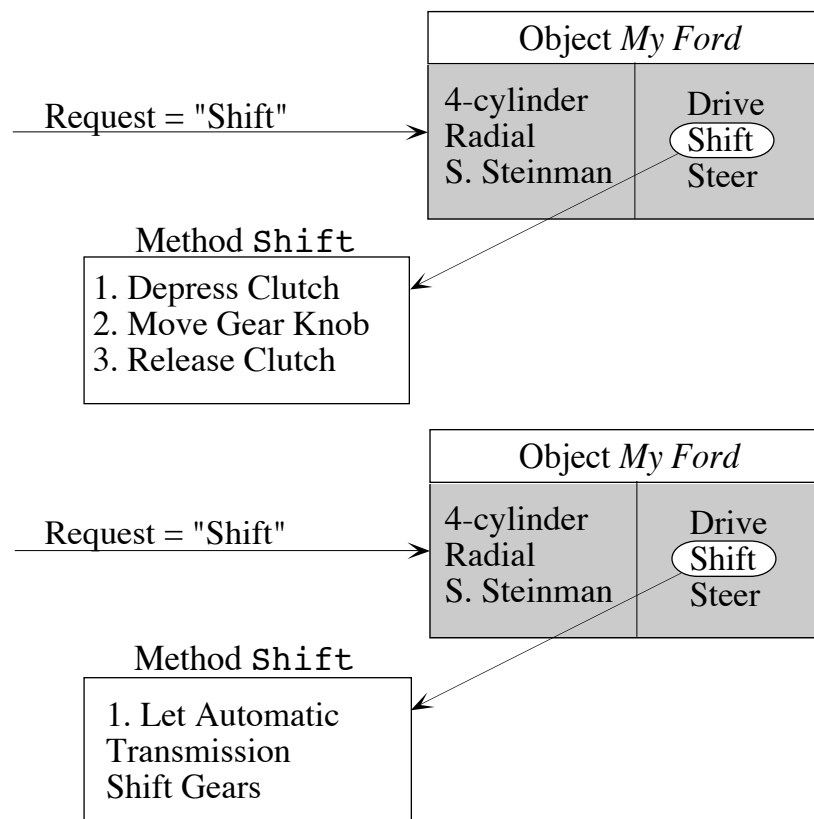


Figure 9.7: Passing a “request” to an object to ask the object to perform an action

The *sender* of the request *doesn't* have to know *how* the receiving object actually *carries out* the action -- it just has to know that the action is carried out. So, in the figure below, the program using the **My Ford** object doesn't need to know *how* the gear is shifted when the request to **Shift** is sent to the **My Ford** object. The program only needs to know that the process of shifting gears *gets done*.

Requesting actions from an object via the class interface has an important benefit -- the ability to *change* a class' implementation details without affecting the programs that use the class. Let's assume our **Car** class contains a class method called **Shift**. Because the details of the **Car** class method **Shift** are “hidden” from the user of the **Car** class, we may change *how* the **Shift** method works (the code within the **Shift** method) with no adverse effects to the rest of the program. For example, whether our **Car** shifts manually or automatically doesn't matter to the program. All the program cares about is that the shifting *is* carried out (see Figure 9.8).

**Figure 9.8: Information hiding in a class allows experimentation with the precise steps taken to perform an action**

This allows us to experiment with different ways of solving problems -- that is, *prototype* our class -- without “breaking” our programs. So long as the request to execute

a method that is sent to the class' object is kept the same, exactly *how* the request is handled by that class method does not matter to the program.

As another example, let's say we want to implement a drawing program. We can specify the shapes that will be drawn as different classes such as **Triangle** or **Circle** or **Square**. The internal details of these classes are left up to us to write in any way we choose so long as it gets the job done. We can represent the shapes in different ways; for example, we can use Euclidian coordinates (x,y pairs) or polar coordinates (magnitude and angle). Each shape can implement its drawing class methods and attributes in either Euclidian or polar format. When we send a request to the shape to "draw itself", it will do so using whichever representation we've decided to give it. But the *program using these shapes* doesn't need to know which representation the class is using to represent its shape. It just has to know that the shape is to be drawn. We can change which representation we use at our leisure, depending upon which is simpler for us to code for a given application.

Software Reuse -- Inheritance and Subclassing

When we look at a particular breed of cat, we immediately see that it has many of the traits of the cat family in general. At the same time, one particular cat is not *exactly* the same as every other feline. A distinct breed of cat *shares* some of the properties and actions of felines in general, but also adds its own *unique* traits or actions that set it apart from other types of cats. For example, a tiger and a house cat both have sharp teeth, claws and fur, but a house cat purrs and drinks milk, while a tiger roars and hunts gazelles. Both the tiger and the house cat *inherit* the general features of what defines a cat, but add *new* attributes that make them distinct from each other. A tiger or house cat can be thought of as different subtypes or *subclasses* of felines.

Similarly, classes within a computer program may be modified to adapt to new applications via *inheritance* (also called *subclassing*). We can derive new classes (*child classes* or *subclasses*) that are automatically given the attributes and methods of our original class (the *parent class* or *superclass*), then add *new* attributes or methods that will allow the new subclass to do new things. In this manner, we can build new classes that *reuse existing code* that we know already works. If the original parent class worked correctly, we can be assured that the inherited code from that class will also work correctly in its new subclass. We save programming time and effort, as well as debugging headaches, by just reusing the working parts of the superclass. What this means is that when we are faced with a new situation that requires a user-defined data type that is similar *but not exactly the same* as a class we've already written, we don't have to start coding all over again from scratch -- we simply build a new data type by adding new capabilities to meet the new program requirements to the existing code from its parent class.

For example, we may create subclasses of our **Car** class that serve other functions, like a **Limousine** class, shown in Figure 9.9. The **Limousine** *inherits* the traits

and actions of a **Car**, since it's just a type of **Car**. The **Limousine** will still accelerate, brake, turn or shift like a **Car**, since it still *is* a **Car**. But the **Limousine** will also have different *traits* than will a **Car**, such as a **chauffeur**, a **wet bar** and a **television**, and may also have different *actions* than does the generic **Car**, such as **Serve Drinks**. That is, while the **Limousine** *shares* the attributes and methods of a **Car**, it will also possess *new attributes and methods of its own*. The time needed to write the **Limousine** class is minimal since most of its code was already written for the **Car** class.

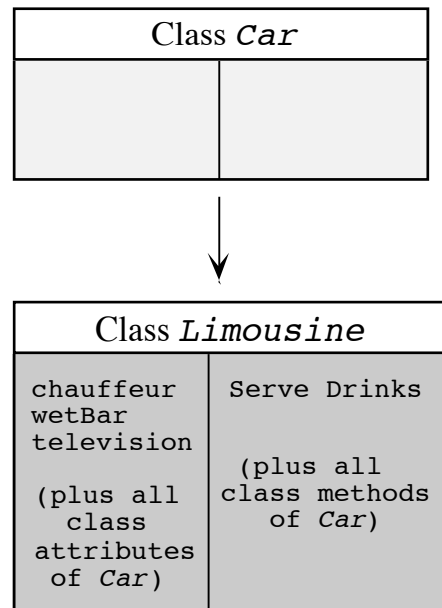


Figure 9.9: Subclassing a class to form a new class

Subclasses are created for two reasons. First, as we've already seen, they can change the behavior of a parent class by *specializing* its behavior with new class methods. The **Limousine** was an example of this kind of subclass. The **Limousine** is a specialized type of **Car** and performs more specialized actions than does a **Car**. The **Limousine** does things that the **Car** cannot.

Subclasses can also add missing functionality to parent classes that are only partially defined -- in other words, the parent class begins the definition of a *set* of new data types and each subclass completes the specification of one new data type. This is a perfectly legal way of writing a special type of parent class called an *abstract superclass*. An abstract superclass is never used by itself -- its sole purpose is to provide common actions to a set of subclasses. An example of this would be an *array*, which is a *fixed-size* list of items. Although we can place some common actions to be performed on the array in the parent class (such as rearranging the array), we can't *fully* complete the code for the array unless we know *what type of data* we'll be placing in the array -- will the array contain characters, integers or real numbers? Obviously, if we specified the type of data in the parent class, the array would store *only that type of data*, reducing the flexibility of the class. Therefore, we make the parent class **Array** an abstract superclass, leaving its

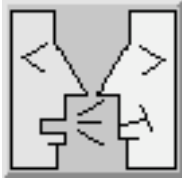
definition incomplete. We then subclass **Array** to define arrays for specific types of data such as **IntegerArray**, **NumberArray** or **CharacterArray**. We'd fill in the data type-specific code in the subclasses.

The additional methods in the subclass can not only add new features to the subclass, but they can be used to *remove* features of the parent class that are no longer needed. So, for example, if a **Limousine** no longer needed to shift gears at all, we could simply write a new **Shift** method for the **Limousine** that was empty. The **Limousine** therefore could not accidentally perform a gear-shifting operation.

Polymorphism

Another powerful feature of classes is *polymorphism*. Let's say you've created some classes to represent geometric shapes for a drawing program. Starting with a general **Shape** class, you could create subclasses such as **Circle** or **Square**. When you wish to draw shapes on the screen, wouldn't it be nice if we could just ask each shape to draw itself without having to know in advance *which* shape object you're sending the request to draw? We could just send the *same request* ("**Draw**") to either class with the same result -- the shape would draw itself correctly. A **Circle** would "know" to draw itself as a round shape, and a **Square** would "know" to draw itself as a four-cornered shape.

The *same Draw* request would mean *different* things, depending upon which type of shape we wanted to draw. Polymorphism is the ability to send the same request to different objects and have each perform its own object-specific actions correctly. It is accomplished by providing class methods with the *same name* in a parent class and its child classes (in this case, **Draw**) whose code differs in each class (see Figure 9.10). The **Draw** method in the **Circle** or **Square** subclasses *overrides* the code of the **Draw** method in the **Shape** superclass, so that while the **Draw** class method performs the same *action* in each class -- drawing -- the *precise steps* required to do the drawing will differ for the **Circle** or **Square**.

**By The Way...**

The C++ programming language distinguishes between *overloading* and *overriding*. *Overloading* is a general term used to define the process by which one class method (or function in C++ parlance) can be called in place of another class method with the same name. In C++, however, it is used specifically to denote the calling of one class method by another of the same name *in the same class*. For example, two class methods named **Multiply** might exist within the same **Complex** number class -- one that multiplies a complex number by another complex number, and the other that multiplies a complex number by a floating-point value. In C++, the precise **Multiply** class method called would be determined by the input parameters fed into the method -- two complex numbers or one complex number and a floating-point number. One **Multiply** method *overloads* the other. Prograph does not have such a mechanism.

Overriding, on the other hand, is the calling of a method in a subclass of a class rather than the method of the same name in the superclass. For example, in the **Shape** example given above, requesting that a **Circle** execute its **Draw** method means that its superclass' **Draw** method (in the **Shape** class) will not be executed. The subclass' method *overrides* that of the superclass. In Prograph, overriding is permitted, and forms a major mechanism by which subclasses can have different actions than their parent classes.

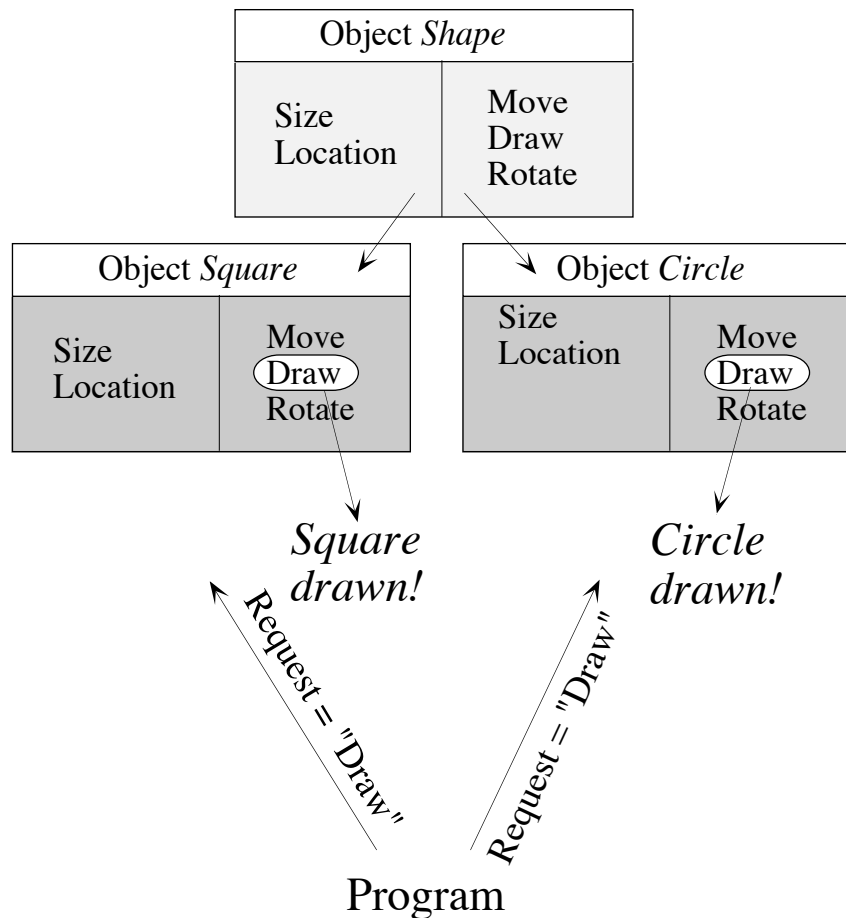


Figure 9.10: Polymorphism -- sending the same request to more than one class

Until we discussed polymorphism, when requests were sent to objects to perform a specific action, the programmer knew when writing the code which objects would be sent the requests and which class methods would perform the actions requested. The Prograph interpreter or compiler would also “know”. This is known as *early or static binding*. Polymorphism, on the other hand, accounts for situations where the programmer, interpreter or compiler *doesn't* know in advance which object will be sent the request -- it could be any one of the subclasses sharing the same class method name. This situation is known as *late or dynamic binding*.

Polymorphism is especially helpful when we deal with *lists* of objects. If our drawing program had a *list of shape objects* that would all be drawn on the screen, we could just *iterate through the shape list*, sending *each shape object in the list* the identical **Draw** request. The request would be given to *several different classes* on each iteration through the list. The list iteration code would draw all of the shapes on the screen properly with the same **Draw** request since each shape in the list would “know” how to draw itself on the screen. The program would “blindly” call the **Draw** method of each of these shape objects without knowing which type of object was being sent the request.

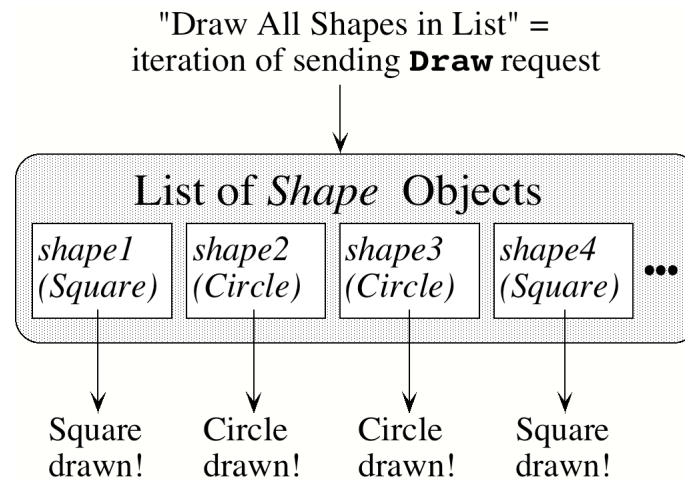
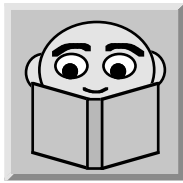


Figure 9.11: Using polymorphism to process a list of objects

Deciding What the Classes and Objects Should Be

We now know that programs can be organized as sets of objects. But how do we know just *what* those objects should be? How do we go about designing an object-oriented program? Let's look at some basic principles of object-oriented design (OOD).



**For More
Information...**

The focus of this book is on learning visual programming using Prograph rather than generic object-oriented programming, and space prevents an in-depth discussion of program design considerations. For more thorough guidelines for applying object-oriented design, see the reference list of OOP books in Chapter 18.

English Descriptions

The first step in designing classes and objects is simply to write down in plain English a thorough description of the problem that you want your program to solve, in as much detail as possible. This description of what the program is required to do is called the *program specifications*. Next, look at the description you've written and make a list of the *nouns* and *verbs* in each sentences of the description. The *nouns* correspond roughly to the *classes and objects* you need to create. The *verbs* denote the *actions* you need the classes to do -- in other words, the *class methods*. While real-world objects named by nouns, such as *dogs*, *trees* or *floppy disks* can be represented as classes, remember that *imaginary things* -- *concepts* -- can also be classes if they can be described concretely. So, for example, a *unicorn* could be a class, even though unicorns don't actually exist. A *complex number* can be a class even though it is merely a mathematical concept. Verbs are *actions* of the objects. If a complex number can be added, subtracted, multiplied or

divided, these will form some of the class methods for the complex number class -- **Add**, **Subtract**, **Multiply** and **Divide**.

Let's return to our racing simulation program. A first attempt at describing the program might read as follows: "A car will be driven by a driver. The driver must avoid hitting other cars and all obstacles." Granted that this isn't a very good description of a complete program, since it lacks many details of actions that the car and driver can take, but let's start with it. What would our classes be? The *nouns* in the sentences are *car*, *driver* and *obstacle*. These would form our initial candidates for objects -- **Car**, **Driver** and **Obstacle**.

The next step is to refine these initial choices for classes. We can ask ourselves whether all of these classes are really necessary, and whether each potential class is clearly defined. How about our choice of **Obstacle**? What *is* an obstacle? An obstacle could be many things, such as a wall or another car. The concept of an obstacle therefore isn't very well defined. We should probably eliminate the **Obstacle** class, or rewrite our program description to better define what obstacles our car should avoid, then make these into classes, such as **Wall** or **Pedestrian**.

Once the classes are defined, we then determine what actions the classes should be able to do. The action verbs in our program description correspond to *drive* (**Car**), *avoid hitting* and *driven by* (**Driver**). These actions are still pretty general at this point, and could form some *high-level* class methods -- **Drive** (classes **Car** and **Driver**), **Avoid Collision** (class **Driver**), etc. But to be able to perform these actions, we must define some more specific, simpler actions. For example, what does a **Car** "*driving*" involve? The **Car** must *accelerate*, *brake*, *turn* and *shift*. These simpler actions will become additional class methods -- **Accelerate**, **Brake**, **Turn** and **Shift** -- that will themselves be called by the general **Drive** method of **Car**. These simpler methods will be easier for us to write than the less-specific **Drive** method. In general, it's a good idea to write both general-purpose and narrowly-focused methods. The general-purpose methods get the larger task done, and the narrowly-focused ones help you write simpler code. This sounds a bit like top-down procedural code, doesn't it? It is, except that here we try to break down general object-oriented class methods into more specific object-oriented class methods.

Another advantage of including smaller, narrowly-defined class functions is that they can be "reused" within the class itself. For example, how would a **Driver** "*drive*"? The **Driver** would try to *avoid collisions*, *watch for other Cars*, and *get his or her own Car to accelerate, turn, brake or shift*. Each of these tasks is a method for the **Driver** class -- **Watch Cars**, **Accelerate**, **Turn** and **Brake**. Now what about the task of "*avoiding collisions*"? This involves some of the same tasks -- **Watch Cars**, **Turn** and **Brake**. We may call these same methods from the **Avoid Collision** method.

What should you look for as good candidates for classes? Budd (see references in Chapter 18) describes four major categories of classes: (1) *Data Managers*, *Data* and

State classes are those that *maintain* data or the state of an operation. Our **Car** class would be an object of this type, since it maintains *data* about the owner of the car and the type of car parts, and maintains the current *state* of the car, such as its current speed and direction. (2) *Data Sinks* or *Data Source* classes generate data or modify it. A random-number generator class would be a type of data source. (3) *View* or *Observer* classes are used to display data on a computer screen. These are representations of how our other objects should appear on the screen if they were to be drawn. So, for example, we could create another class called **CarView** that would contain the representation and methods for drawing our **Car** on the screen. It's a good idea to separate your data type from the class that determines how that data will be drawn. This way, you can more easily modify your program to work on different graphics devices, or port the program to other computers. (4) *Facilitator* or *Helper* classes exist solely to help other classes do their own functions. A common example of a helper class is a *sorted list* class.

Sometimes as we write descriptions of programs, the nouns are accompanied by *adjectives* such as “a *text* file” or “a *rounded* rectangle”. Watch for these! These may be a clue to the need for *another* type of object. Instead of using a class **Rectangle**, you might want to use a more specific class called **RoundedRectangle**. We'll talk about how to do this when we discuss how to design *subclasses*.

As you continue to design your program, you will probably find that your program specifications become more focused, and some of the classes that you initially planned weren't as good as you first thought they'd be. Either these initial classes wind up to be too hard to design clearly, or the implementation details of their class methods are too difficult to write, or the objects don't really meet the specifications of the program. Don't be discouraged by this. Program design can be a *continuous, evolving process*. As the program specifications are refined, you'll refine your notions about what your objects should be and what they should do. This is called *stepwise refinement*. OOP and Prograph both support stepwise changes in classes and programs very well. As you gain more experience using OOD, you may be able to choose better initial classes, but your programs will still almost always go through a stepwise refinement process.

Exercise 9.2:

Write a complete description of a program that will catalog the files stored on your computer's hard disk and floppy disks. Find the tentative classes that this program can use and the attributes and methods of the classes.

Class-Responsibility-Collaborator (CRC) Cards

Once you have *some* idea of *what* your classes should be, you can progress to a more exact design technique for constructing the inner workings of the classes. One widely adopted technique uses *Class-Responsibility-Collaborator (CRC) cards*, shown in Figure 9.12. The CRC card derives its name from the fact that its information was originally written on an index card.

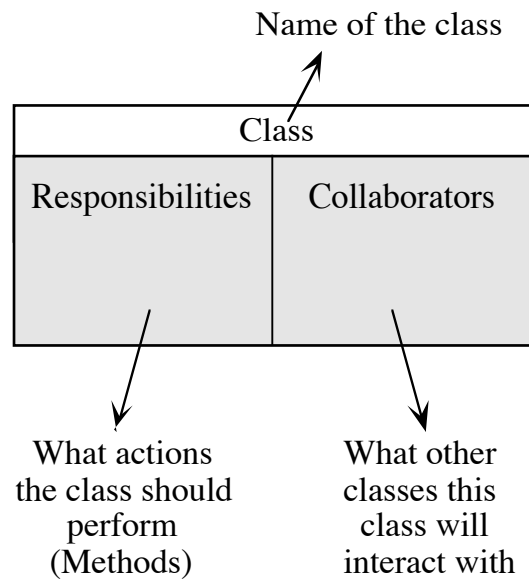


Figure 9.12: CRC card specification of a class

On the top of the CRC index card, we write the *name of the class* we are designing. On the left side of the card, we list the *responsibilities* of the class -- *the tasks that the class is supposed to do*. Each of these responsibilities will eventually be translated into a *class method* or *group of class methods*. On the right side, we list the *collaborators* of the class. Collaborators are *all of the other classes upon which this class depends* to get its job done. For example, our **Car** class is dependent upon the **Driver** class. These two classes must *intercommunicate* if the **Car** is to be able to drive. The **Driver** class is therefore a *collaborator* of the **Car** class. A CRC card for our **Car** class would look something like Figure 9.13.

Class <i>Car</i>	
Responsibilities	Collaborators
Accelerate Brake Shift StartIgnition Steer	Driver TrafficSignal

Figure 9.13: CRC card for the Car class

This CRC card is a concise representation of what tasks the **Car** class needs to do, and what other classes it must work with to do these tasks. The list of responsibilities is kept as short as possible so that the class does not become a huge collection of methods, which would defeat the purpose of building tightly-knit classes. A good rule of thumb is that the list of responsibilities of a potential class should not be so long that it can't fit on

one side of an index card. If a potential class has too many responsibilities, you should think about whether that class should be split into two classes with fewer chores performed by each.

Why do we need collaborators at all? If a single class did everything for your program all by itself, it would be extremely hard to write and modify. For example, try writing a class called **Government** that carried out all of the duties of a country's government. This would be a monumental chore! How would you even begin to do it? It makes more sense to split apart the duties of a complicated class so that they're carried out by a handful of simpler and smaller classes that *interact* with each other. The huge number of tasks that the **Government** class would need to do would be more easily carried out by **Military**, **Judicial**, **Legislative** and **Executive** classes, to name a few. Each of these classes would have to handle a narrower range of responsibilities, making the writing and testing of each class easier.

Once you've fleshed out your potential classes in this manner, you may wish to use the back of the index card to start listing the internal *attributes* of each class. Once again, try to keep your list of data items as short as possible without compromising the function of the class.

Exercise 9.3:

Using 3" by 5" index cards, construct CRC card diagrams for the classes you chose to use in Exercise 9.2, modifying the class definitions if necessary.

Changing a Class' Behavior by Subclassing

Earlier, we touched upon a means of reusing our existing classes when creating new classes. This process was called *subclassing*. The original class is called a *superclass* and the new, more specialized classes are called *subclasses*. After designing initial classes with the CRC card method, we may start thinking about *class hierarchies* -- families of classes that are designed to be as reusable as possible.

Finding Subclasses

When we started designing classes for our programs, we first looked at nouns contained in the sentences of our program description. Now we need to look at those nouns again so we may refine our design. In our driving simulation example, we settled on a *car* as a potential class. Now we must ask ourselves if a **Car** would really be the simplest object we could construct. Would a **Car** class be very *reusable*? Would it allow for the minimum amount of code rewriting if we wanted to make new *subclasses* based upon the **Car**?

A car is already a fairly specialized thing. A car is just one type of *vehicle*. Vehicles encompass all wheeled transportation devices, including cars, trucks, vans,

motorcycles, etc. So we'd be better off starting with a more general **Vehicle** class rather than a **Car**. **Vehicle** would be a better starting class since we could easily subclass it to make other types of conveyances besides cars. A car is a *type* of vehicle, but so is a truck, a minivan, a motorcycle or a bicycle. In fact, this is the basic rule of finding subclasses to create. It's called the “*is-a*” relationship. If a car *is a* vehicle, then class **Car** should be a *subclass* of class **Vehicle**.

We can carry this one step further. Vehicles are themselves a type of *transporter* - they are wheeled transporters that transport people and goods. Other types of transporters that *don't* have wheels, but still carry things, are boats and planes. We could create a **Transporter** class, then subclass it to make *any* type of transporter or vehicle. All actions and attributes that are common to *all transporters* would be written once for the **Transporter** class, then *all of these possible subclasses* would simply reuse them. We might then create special types of transporters for use on the ground (**Vehicle**), in the air (**AeroTransporter**, for want of a better term) and in the water (**AquaTransporter**). These would then be subclassed again to create specific classes such as **Car**, **Boat**, **Plane**, etc.

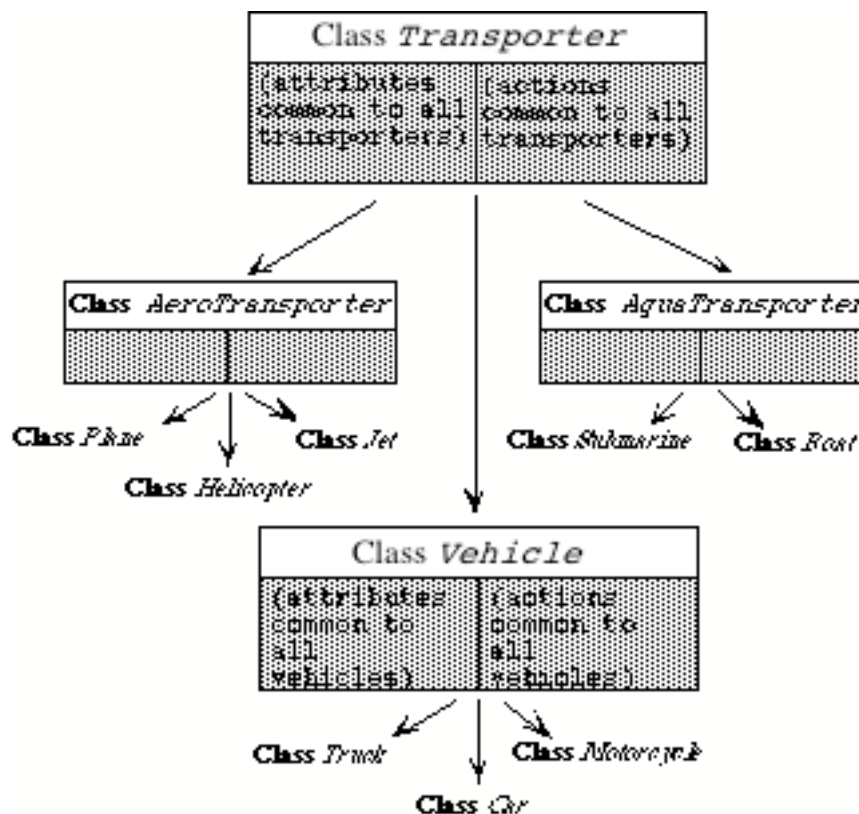


Figure 9.14: A class inheritance tree

Just what have we been doing so far? We've *looked for the common behavior of several classes* and “factored it out” into a more general *superclass*. The common behavior is carried out by one set of common attributes and class methods. More

specialized classes built upon the superclass would only need new attributes and methods to account for any changes in their behavior relative to the superclass. If your program might use two or more classes that are very similar and that have a lot of similar methods, consider using a more *general* class as the *superclass* of the two classes and moving their *common code* to the superclass. In fact, even if you don't need a more general superclass for the program you're writing *right now*, you might create it anyway so it will be available for you to use in later programs. Exactly how general you make your top-most class depends upon how much you expect to reuse the classes in the present program and, in more realistic terms, how much of a rush you're in to get the @#\$%&* program running!

Exercise 9.4:

Design a class hierarchy for warm-blooded animals such as dogs, lions, monkeys, etc.

Try to start with general classes that can encompass the actions of several potential classes. These will form *abstract superclasses* that you will then subclass. Remember that abstract superclasses themselves are *never* used to create objects -- only the *subclasses* that inherit from the abstract superclass are used to create objects. For example, a **Shape** would be an abstract superclass. You'd never make an object from a generic shape, since it wouldn't be useful. You'd make objects from *subclasses* of **Shape**, such as **Circle** or **Square**. The **Shape** class is used solely for the purpose of subclassing and code reuse. All behavior that's common to *all* shapes is implemented in the **Shape** class. When a *specific type of shape* is needed, the subclasses of **Shape** are used to create objects with more specific behavior. The benefit of the abstract superclass is that it can be used over and over again to create other subclasses easily. Since you've already implemented the behavior that's common to all shapes, all you have to write is what makes a new shape different from a general shape.

As another tool for finding common superclasses, Wirfs-Brock, Wilkerson and Wiener suggest drawing Venn diagrams. If the sets defined by two of your potential classes overlap, then they must share a common superclass.

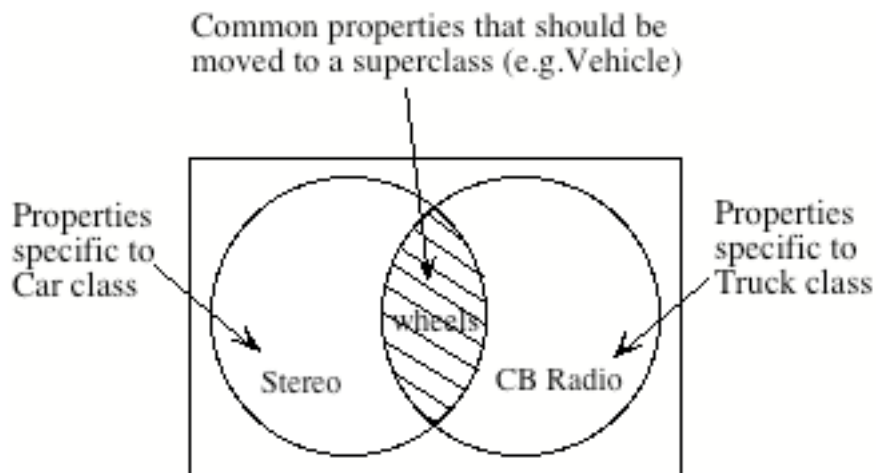
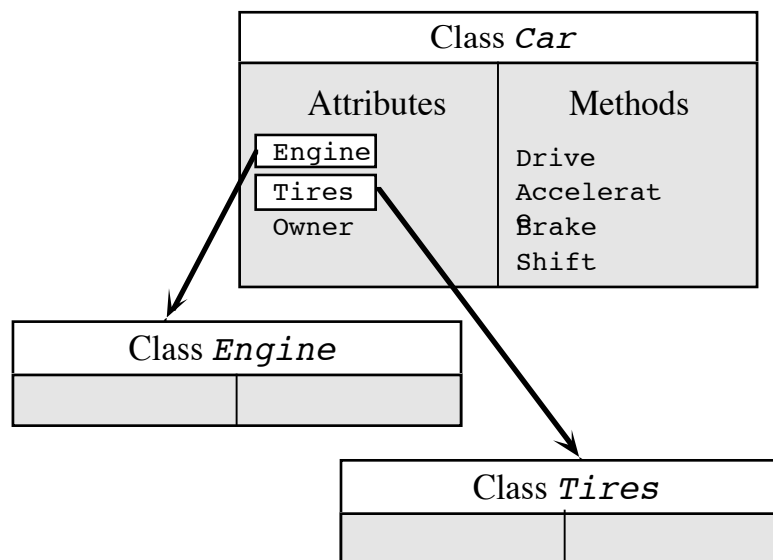


Figure 9.15: Determining a candidate for a superclass via a Venn diagram***Software Reuse -- Extending a Class with Composition (Incorporating Instances of Other Classes)***

Subclassing or inheritance was defined as satisfying the “is-a” relationship. There is a second type of relationship that objects can have, which defines another way of reusing code. This relationship is called the “has-a” relationship, and the class design technique is called *composition*. If two classes can be related by a “has-a” relationship, then one class should *contain* an instance (object) of the other object. For example, in our **Car** class, a **Car** *has* an engine, *has* tires and *has* an owner. The **Car** class should therefore *contain* an object of the **Engine** class or an **Owner** class (if you wish these data to be represented by objects) included in its *attributes*. Remember that classes can contain any data element type in its attributes, even instances of *other classes*.

The objects contained inside another object can be accessed by the containing object. This is another form of *collaboration* between classes. It can be quite helpful in situations where one class performs duties that are also useful for other classes. If we include an instance of **Engine** inside our **Car** to become one of a **Car** object’s attributes, the **Car** will be able to get its **Engine** to do things without needing to know how the **Engine** actually works. This is therefore also another form of information hiding -- taking advantage of classes without needing to know how they accomplish their tasks.

**Figure 9.16: Composition -- placing instances (objects) within classes**

Another example of the use of composition is the design of a **Rectangle** class. A **Rectangle** is composed of four *lines*, which can each be defined by a **Line** object. An object created from the **Rectangle** class could therefore itself contain two objects of the

Line class. Lines themselves are each specified by two *points* -- the endpoints of the line. So the Line class could also contain two instances of a **Point** class.

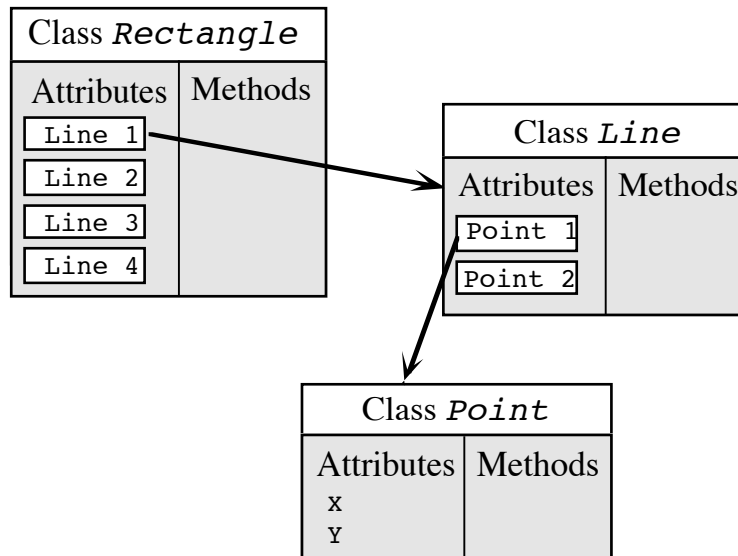


Figure 9.17: The use of composition for graphical constructs

Exercise 9.5:

Design a Human Being class with composition, using individual classes for body parts.

Summary

Object-oriented programming (OOP) helps to reduce some of the problems inherent in large programming projects. The proper use of OOP makes programs more easily maintained, modified, extended and debugged.

The most important construct of object-oriented programming is the *class*, which models real-world concepts or objects in software.

- Classes are composed of both data (*attributes*) and code (*class methods*).
- The attributes of a class represent both the class' *traits*, what makes this class different from other classes, and its current *state*.
- The class methods of a class delineate what *actions* the class can carry out.
- Classes are templates for the creation of individual instances, called *objects*, which are typically used as variables in programs.
- In the Prograph language, attributes can be further subdivided as either *instance attributes*, or *class attributes*. Instance attributes have values that

differ from one object to the next, and differentiate each particular instance of the class. Class attributes have the same value for every object created from that class, and serve as shared data.

- “Communication” with objects is accomplished by *sending requests to the object to perform actions*. The object executes one of its class methods to satisfy the request. Our program can send a request for action to an object, or one object can send a request to another object.

Object-oriented programming is defined by three major characteristics:

- *Data abstraction* is the ability to create *new data types* with a class. *Encapsulation* is use of these new data types in programs without needing to know how their inner workings are defined.
- *Inheritance* or *subclassing* is the creation of new classes (*subclasses*) that reuse portions of the original class (*superclass*), but also add new features to extend or their usefulness or make the class more specific with a minimum of reprogramming.
- *Polymorphism* is the ability to request the same action from different objects, and have these different objects automatically perform these actions in slightly different ways that are specific to their own classes. Polymorphism is one way to accomplish *dynamic binding*, where the application code does not know which object’s method is being called until run-time -- the same request can be sent to any one of the subclasses of a given parent class without the calling code “knowing” in advance which subclass receives the request.

In this chapter, we described methods for determining what the classes and objects of your programs should be, as well as how and when you should be using *inheritance* or *composition* (instances of classes embedded within other objects) to add new features to classes.

- Inheritance is used to satisfy an “*is-a*” *relationship* between classes, such as a car *is a* type of vehicle. Therefore, a car should be a subclass of a vehicle.
- *Composition* is the placement of an object of one class within another class, and is used to satisfy an “*has-a*” *relationship* between classes, such as a car *has an* engine. An engine should be an instance of an **Engine** class placed within an instance of a **Car** class.

We’ve covered quite a lot of ground in this chapter. At this point, OOP is just a theoretical construct to you. Now we’re going to reinforce what you’ve learned in this chapter by showing you how to *apply* object-oriented programming principles to Prograph programs.