CHAPTER 10

Classes and Object-Oriented Programming

TOPICS

10.1 Procedural and Object-Oriented Programming

CONCEPT: **Procedural programming is a method of writing software. It is a programming practice centered on the procedures or actions that take place in a program. Object-oriented programming is centered on objects. Objects are created from abstract data types that encapsulate data and functions together.**

There are primarily two methods of programming in use today: procedural and object-oriented. The earliest programming languages were procedural, meaning a program was made of one or more procedures. You can think of a *procedure* simply as a function that performs a specific task such as gathering input from the user, performing calculations, reading or writing files, displaying output, and so on. The programs that you have written so far have been procedural in nature.

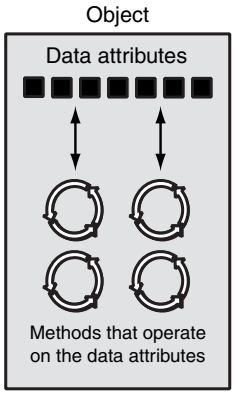
Typically, procedures operate on data items that are separate from the procedures. In a procedural program, the data items are commonly passed from one procedure to another. As you might imagine, the focus of procedural programming is on the creation of procedures that operate on the program's data. The separation of data and the code that operates on the data can lead to problems, however, as the program becomes larger and more complex.

For example, suppose you are part of a programming team that has written an extensive customer database program. The program was initially designed so a customer's name,

address, and phone number were referenced by three variables. Your job was to design several functions that accept those three variables as arguments and perform operations on them. The software has been operating successfully for some time, but your team has been asked to update it by adding several new features. During the revision process, the senior programmer informs you that the customer's name, address, and phone number will no longer be stored in variables. Instead, they will be stored in a list. This means you will have to modify all of the functions that you have designed so they accept and work with a list instead of the three variables. Making these extensive modifications not only is a great deal of work, but also opens the opportunity for errors to appear in your code.

Whereas procedural programming is centered on creating procedures (functions), *object-oriented programming* (*OOP*) is centered on creating objects. An *object* is a software entity that contains both data and procedures. The data contained in an object is known as the object's *data attributes*. An object's data attributes are simply variables that reference data. The procedures that an object performs are known as *methods*. An object's methods are functions that perform operations on the object's data attributes. The object is, conceptually, a self-contained unit that consists of data attributes and methods that operate on the data attributes. This is illustrated in Figure 10-1.

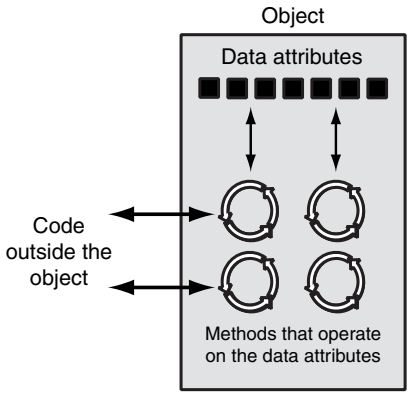
Figure 10-1 An object contains data attributes and methods



OOP addresses the problem of code and data separation through encapsulation and data hiding. *Encapsulation* refers to the combining of data and code into a single object. *Data hiding* refers to an object's ability to hide its data attributes from code that is outside the object. Only the object's methods may directly access and make changes to the object's data attributes.

An object typically hides its data, but allows outside code to access its methods. As shown in Figure 10-2, the object's methods provide programming statements outside the object with indirect access to the object's data attributes.

Figure 10-2 Code outside the object interacts with the object's methods



When an object's data attributes are hidden from outside code, and access to the data attributes is restricted to the object's methods, the data attributes are protected from accidental corruption. In addition, the code outside the object does not need to know about the format or internal structure of the object's data. The code only needs to interact with the object's methods. When a programmer changes the structure of an object's internal data attributes, he or she also modifies the object's methods so they may properly operate on the data. The way in which outside code interacts with the methods, however, does not change.

Object Reusability

In addition to solving the problems of code and data separation, the use of OOP has also been encouraged by the trend of *object reusability*. An object is not a stand-alone program, but is used by programs that need its services. For example, Sharon is a programmer who has developed a set of objects for rendering 3D images. She is a math whiz and knows a lot about computer graphics, so her objects are coded to perform all of the necessary 3D mathematical operations and handle the computer's video hardware. Tom, who is writing a program for an architectural firm, needs his application to display 3D images of buildings. Because he is working under a tight deadline and does not possess a great deal of knowledge about computer graphics, he can use Sharon's objects to perform the 3D rendering (for a small fee, of course!).

An Everyday Example of an Object

Imagine that your alarm clock is actually a software object. If it were, it would have the following data attributes:

- `current_second` (a value in the range of 0–59)
- `current_minute` (a value in the range of 0–59)
- `current_hour` (a value in the range of 1–12)
- `alarm_time` (a valid hour and minute)
- `alarm_is_set` (True or False)

As you can see, the data attributes are merely values that define the *state* in which the alarm clock is currently. You, the user of the alarm clock object, cannot directly manipulate these data attributes because they are *private*. To change a data attribute's value, you must use one of the object's methods. The following are some of the alarm clock object's methods:

- `set_time`
- `set_alarm_time`
- `set_alarm_on`
- `set_alarm_off`

Each method manipulates one or more of the data attributes. For example, the `set_time` method allows you to set the alarm clock's time. You activate the method by pressing a button on top of the clock. By using another button, you can activate the `set_alarm_time` method.

In addition, another button allows you to execute the `set_alarm_on` and `set_alarm_off` methods. Notice all of these methods can be activated by you, who are outside the alarm clock. Methods that can be accessed by entities outside the object are known as *public methods*.

The alarm clock also has *private methods,* which are part of the object's private, internal workings. External entities (such as you, the user of the alarm clock) do not have direct access to the alarm clock's private methods. The object is designed to execute these methods automatically and hide the details from you. The following are the alarm clock object's private methods:

- `increment_current_second`
- `increment_current_minute`
- `increment_current_hour`
- `sound_alarm`

Every second the `increment_current_second` method executes. This changes the value of the `current_second` data attribute. If the `current_second` data attribute is set to 59 when this method executes, the method is programmed to reset `current_second` to 0, and then cause the `increment_current_minute` method to execute. This method adds 1 to the `current_minute` data attribute, unless it is set to 59. In that case, it resets `current_minute` to 0 and causes the `increment_current_hour` method to execute. The `increment_current_minute` method compares the new time to the `alarm_time`. If the two times match and the alarm is turned on, the `sound_alarm` method is executed.

Checkpoint

10.1 What is an object?

10.2 What is encapsulation?

10.3 Why is an object's internal data usually hidden from outside code?

10.4 What are public methods? What are private methods?

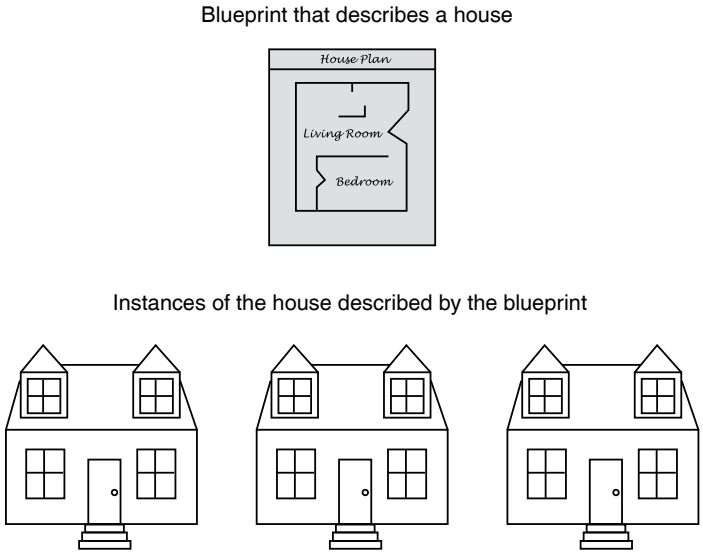
10.2 Classes

CONCEPT: A class is code that specifies the data attributes and methods for a particular type of object.

VideoNote
Classes and Objects

Now, let's discuss how objects are created in software. Before an object can be created, it must be designed by a programmer. The programmer determines the data attributes and methods that are necessary, then creates a *class*. A class is code that specifies the data attributes and methods of a particular type of object. Think of a class as a "blueprint" from which objects may be created. It serves a similar purpose as the blueprint for a house. The blueprint itself is not a house, but is a detailed description of a house. When we use the blueprint to build an actual house, we could say we are building an *instance* of the house described by the blueprint. If we so desire, we can build several identical houses from the same blueprint. Each house is a separate instance of the house described by the blueprint. This idea is illustrated in Figure 10-3.

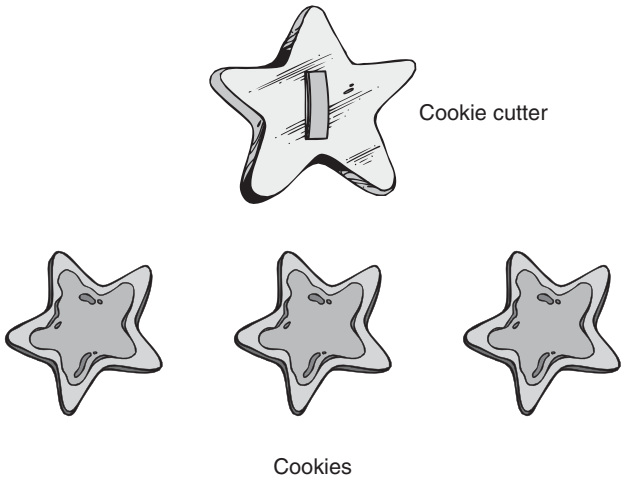
Figure 10-3 A blueprint and houses built from the blueprint



Another way of thinking about the difference between a class and an object is to think of the difference between a cookie cutter and a cookie. While a cookie cutter itself is not a cookie, it describes a cookie. The cookie cutter can be used to make several cookies, as shown in Figure 10-4. Think of a class as a cookie cutter, and the objects created from the class as cookies.

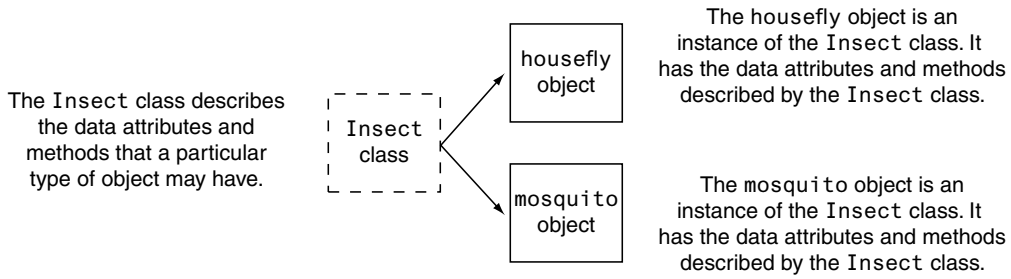
So, a class is a description of an object's characteristics. When the program is running, it can use the class to create, in memory, as many objects of a specific type as needed. Each object that is created from a class is called an *instance* of the class.

Figure 10-4 The cookie cutter metaphor



For example, Jessica is an entomologist (someone who studies insects), and she also enjoys writing computer programs. She designs a program to catalog different types of insects. As part of the program, she creates a class named `Insect`, which specifies characteristics that are common to all types of insects. The `Insect` class is a specification from which objects may be created. Next, she writes programming statements that create an object named `housefly`, which is an instance of the `Insect` class. The `housefly` object is an entity that occupies computer memory and stores data about a housefly. It has the data attributes and methods specified by the `Insect` class. Then she writes programming statements that create an object named `mosquito`. The `mosquito` object is also an instance of the `Insect` class. It has its own area in memory and stores data about a mosquito. Although the `housefly` and `mosquito` objects are separate entities in the computer's memory, they were both created from the `Insect` class. This means that each of the objects has the data attributes and methods described by the `Insect` class. This is illustrated in Figure 10-5.

Figure 10-5 The `housefly` and `mosquito` objects are instances of the `Insect` class



Class Definitions

To create a class, you write a *class definition*. A class definition is a set of statements that define a class's methods and data attributes. Let's look at a simple example. Suppose we are writing a program to simulate the tossing of a coin. In the program, we need to repeatedly