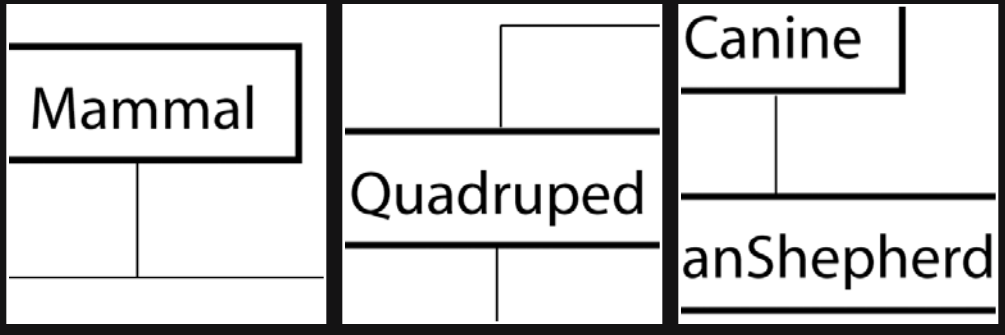
8 OBJECT-ORIENTED PROGRAMMING



Object-oriented programming (OOP) is a deep and complex subject, and a detailed discussion of it is beyond the scope of this book. However, in this chapter I will give you a solid grounding in OOP to get you started. While OOP is challenging to start with, it is fundamental to Java and therefore Processing, and it is very rewarding once you get the hang of it, so don't despair if you find this chapter difficult to follow. As you work through the rest of the book, it will help you to keep returning to this chapter to look up OOP concepts as you meet them in all the applied examples you come across. It will get easier as you practice, I promise.

A new way of programming?

The answer to this question is both yes and no. While very different from procedural programming, OOP can at the same time be thought of as a natural extension of it. In procedural programming, you use functions to modularize code, adding efficiency, organization, and some portability. Classes, the main building blocks of OOP, work similarly. However, OOP takes the modularization of code a couple of magnitudes further. Functions are like data processing machines. They input data, perform some computation with/on the data, and spit the data back out to be used somewhere else. Users who utilize a function only need to know what the function does (not how it does it) and what type of data the function is expecting as input. This type of approach is often referred to as "black-box" design. Electrical components work this way (thankfully). Imagine if you had to understand all the details of how your DVD player works just to watch a film. Instead, you just need to know how to plug it in and hit play. Functions and classes both provide their own variations on black-box design; they encapsulate their internal implementation while providing a public interface to utilize the structure.

Classes, like functions, are modular, can process data, and allow code to be organized into logical structures, adding more organization to the coding process. However, classes go much further than functions in their ability to be independent, modular, reusable entities. In addition, classes have built-in variables, called **properties**, and their own functions, called **methods**—these properties and methods can be used beyond anything a function could provide. In OOP, we say that we interact with a class through its public interface, which can be thought of as the portal to communicate with the class's black box. Also, unlike functions—which are singular, static processing units—classes provide customized copies of themselves called **instances**, allowing a single class to have a wide range of unique states or instances. Instances of a class are also referred to as **objects**, which, if it isn't obvious, is where the term object-oriented programming comes from.

For example, if I create a Box class, I can then use the class to create Box instances of varying size, shape, color, and so on. Each of the Box instances would share certain core attributes (properties), but these attributes could each be expressed uniquely. Classes can also extend other classes. For example, I can create a class called Shape, which I can then extend to create `Rectangle`, `Circle`, and `Triangle` classes. Each of these three more-specific classes would share certain common attributes, such as position, size, and color, defined within the common Shape class. Aside from the common attributes, instances of these classes will also have their own unique characteristics, such as a specific plotting algorithm, a radius (for a circle), width/height (for a rectangle), an orientation (for a trian-