CHAPTER

4 Inheritance

AS mentioned in Chapter 3, an important goal of object-oriented programming is code reuse. Just as engineers use components over and over in their designs, programmers should be able to reuse objects rather than repeatedly reimplement them. In an object-oriented programming language, the fundamental mechanism for code reuse is *inheritance*. Inheritance allows us to extend the functionality of an object. In other words, we can create new types with restricted (or extended) properties of the original type, in effect forming a hierarchy of classes. Also, inheritance is the mechanism that Java uses to implement generic methods and classes.

In this chapter, we will see:

- General principles of inheritance, including *polymorphism*
- How inheritance is implemented in Java
- How a collection of classes can be derived from a single abstract class
- The *interface*, which is a special kind of a class
- How Java implements generic programming using inheritance

4.1 What Is Inheritance?

Inheritance is the fundamental object-oriented principle that is used to reuse code among related classes. Inheritance models the *IS-A relationship*. In an IS-A relationship, we say the derived class *is a* (variation of the) base class. For example, a Circle IS-A Shape and a Car IS-A Vehicle. However, an Ellipse IS-NOT-A Circle. Inheritance relationships form *hierarchies*. For instance, we can extend Car to other classes, since a ForeignCar IS-A Car (and pays tariffs) and a DomesticCar IS-A Car (and does not pay tariffs), and so on.

In an *IS-A relationship*, we say the derived class *is a* (variation of the) base class.

Another type of relationship is a *HAS-A* (or IS-COMPOSED-OF) *relationship*. This type of relationship does not possess the properties that would be natural in an inheritance hierarchy. An example of a HAS-A relationship is that a car HAS-A steering wheel. HAS-A relationships should not be modeled by inheritance. Instead, they should use the technique of *composition*, in which the components are simply made private data fields.

In a *HAS-A relationship*, we say the derived class *has a* (instance of the) base class. *Composition* is used to model HAS-A relationships.