Discovering Inheritance, Polymorphism, and Interfaces

An object-based language is a language that encapsulates state and behaviors in objects. Java’s support for encapsulation (discussed in Chapter 3) qualifies it as an object-based language. However, Java is also an object-oriented language because it supports inheritance and polymorphism (as well as encapsulation). (Object-oriented languages are a subset of object-based languages.) In Chapter 4 I introduce you to Java’s language features that support inheritance and polymorphism. Also, I introduce you to interfaces, Java’s ultimate abstract type mechanism.

Building Class Hierarchies

We tend to categorize stuff by saying things like “cars are vehicles” or “savings accounts are bank accounts.” By making these statements, we really are saying (from a software development perspective) that cars inherit vehicular state (e.g., make and color) and behaviors (e.g., park and display mileage) and that savings accounts inherit bank account state (e.g., balance) and behaviors (e.g., deposit and withdraw). Car, vehicle, savings account, and bank account are examples of real-world entity categories, and inheritance is a hierarchical relationship between similar entity categories in which one category inherits state and behaviors from at least one other entity category. Inheriting from a single category is single inheritance, and inheriting from at least two categories is multiple inheritance.

Java supports single inheritance and multiple inheritance to facilitate code reuse—why reinvent the wheel? Java supports single inheritance in a class context in which a class inherits state and behaviors from another class through class extension. Because classes are involved, Java refers to this kind of inheritance as implementation inheritance.

Java also supports single inheritance and multiple inheritance in an interface context in which a class inherits behavior templates from one or more interfaces through interface implementation or in which an interface inherits behavior templates from one or more interfaces through interface extension.
Because interfaces are involved, Java refers to this kind of inheritance as *interface inheritance*. (I discuss interfaces later in this chapter.)

**Note** You reuse code by carefully extending classes, implementing interfaces, and extending interfaces. You start with something that is close to what you want and extend it to meet your goal. You don’t reuse code by simply copying and pasting it. Copying and pasting often results in redundant (i.e., nonreusable) and buggy code.

In this section I introduce you to Java’s support for implementation inheritance by first focusing on class extension. I then introduce you to a special class that sits at the top of Java’s class hierarchy. After introducing you to composition, which is an alternative to implementation inheritance for reusing code, I show you how composition can be used to overcome problems with implementation inheritance.

### Extending Classes

Java provides the reserved word `extends` for specifying a hierarchical relationship between two classes. For example, suppose you have a `Vehicle` class and want to introduce a `Car` class as a kind of `Vehicle`. Listing 4-1 uses `extends` to cement this relationship.

**Listing 4-1. Relating Two Classes via `extends`**

```java
class Vehicle {
    // member declarations
}

class Car extends Vehicle {
    // member declarations
}
```

Listing 4-1 codifies a relationship that is known as an “is-a” relationship: a car is a kind of vehicle. In this relationship, `Vehicle` is known as the *base class*, *parent class*, or *superclass*; and `Car` is known as the *derived class*, *child class*, or *subclass*.

**Caution** You cannot extend a `final` class. For example, if you declared `Vehicle` as `final` class `Vehicle`, the compiler would report an error on encountering class `Car extends Vehicle`. Developers declare their classes `final` when they don’t want these classes to be extended (for security or other reasons).