10.1 Introduction

We continue our study of object-oriented programming by explaining and demonstrating **polymorphism** with inheritance hierarchies. Polymorphism enables you to “program in the *general*” rather than “program in the *specific*.” In particular, polymorphism enables you to write programs that process objects that share the same superclass, either directly or indirectly, as if they were all objects of the superclass; this can simplify programming.

Consider the following example of polymorphism. Suppose we create a program that simulates the movement of several types of animals for a biological study. Classes `Fish`, `Frog` and `Bird` represent the types of animals under investigation. Imagine that each class extends superclass `Animal`, which contains a method `move` and maintains an animal’s current location as *x*-*y* coordinates. Each subclass implements method `move`. Our program maintains an `Animal` array containing references to objects of the various `Animal` subclasses. To simulate the animals’ movements, the program sends each object the *same* message once per second—namely, `move`. Each specific type of `Animal` responds to a `move` message in its own way—a `Fish` might swim three feet, a `Frog` might jump five feet and a `Bird` might fly ten feet. Each object knows how to modify its *x*-*y* coordinates appropriately for its *specific* type of movement. Relying on each object to know how to “do the right thing” (i.e., do what’s appropriate for that type of object) in response to the *same* method call is the key concept of polymorphism. The *same* message (in this case, `move`) sent to a *variety* of objects has *many forms* of results—hence the term polymorphism.

Implementing for Extensibility

With polymorphism, we can design and implement systems that are easily *extensible*—new classes can be added with little or no modification to the general portions of the program, as long as the new classes are part of the inheritance hierarchy that the program processes generically. The new classes simply “plug right in.” The only parts of a program that must be altered are those that require direct knowledge of the new classes that we add to the hierarchy. For example, if we extend class `Animal` to create class `Tortoise` (which might respond to a `move` message by crawling one inch), we need to write only the `Tortoise` class and the part of the simulation that instantiates a `Tortoise` object. The portions of the simulation that tell each `Animal` to move generically can remain the same.

Chapter Overview

First, we discuss common examples of polymorphism. We then provide a simple example demonstrating polymorphic behavior. We use superclass references to manipulate *both* superclass objects and subclass objects polymorphically.

We then present a case study that revisits the employee hierarchy of Section 9.4.5. We develop a simple payroll application that polymorphically calculates the weekly pay of several different types of employees using each employee's `earnings` method. Though the earnings of each type of employee are calculated in a *specific* way, polymorphism allows us to process the employees "in the *general*." In the case study, we enlarge the hierarchy to include two new classes—`SalariedEmployee` (for people paid a fixed weekly salary) and `HourlyEmployee` (for people paid an hourly salary and "time-and-a-half" for overtime). We declare the common functionality for all the classes in the updated hierarchy in an "abstract" `Employee` class from which "concrete" classes `SalariedEmployee`, `HourlyEmployee` and `CommissionEmployee` inherit directly and "concrete" class `BasePlusCommissionEmployee` inherits indirectly. As you'll see, *when we invoke each employee's* `earnings` *method off a superclass* `Employee` *reference (regardless of the employee's type), the correct earnings subclass calculation* *is performed*, due to Java’s built-in polymorphic capabilities.

Programming in the Specific

Occasionally, when performing polymorphic processing, we need to program “in the *specific*.” Our `Employee` case study demonstrates that a program can determine the *type* of an object at *execution time* and act on that object accordingly. In the case study, we’ve decided that `BasePlusCommissionEmployee`s should receive 10% raises on their base salaries. So, we use these capabilities to determine whether a particular employee object *is a* `BasePlus-CommissionEmployee`. If so, we increase that employee’s base salary by 10%.

Interfaces

The chapter continues with an introduction to Java *interfaces*, which are particularly useful for assigning *common* functionality to possibly *unrelated* classes. This allows objects of these classes to be processed polymorphically—objects of classes that **implement** the *same* interface can respond to all of the interface method calls. To demonstrate creating and using interfaces, we modify our payroll application to create a generalized accounts payable application that can calculate payments due for company employees *and* invoice amounts to be billed for purchased goods.