9 Polymorphism

CHAPTER OBJECTIVES

- Define polymorphism and explore its benefits.
- Discuss the concept of dynamic binding.
- Use inheritance to create polymorphic references.
- Explore the purpose and syntax of Java interfaces.
- Use interfaces to create polymorphic references.
- Discuss object-oriented design in the context of polymorphism.

This chapter discusses polymorphism, another fundamental principle of object-oriented software. We first explore the concept of binding and discuss how it is related to polymorphism. Then we look at two distinct ways to implement a polymorphic reference in Java: inheritance and interfaces. Next we explore Java interfaces in general, establishing the similarities between them and abstract classes, and bringing the polymorphism discussion full circle. The chapter concludes with a discussion of the design issues related to polymorphism.

9.1 Dynamic Binding

Often, the type of a reference variable exactly matches the class of the object to which it refers. For example, consider the following reference:

```
ChessPiece bishop;
```

The `bishop` variable may be used to point to an object that is created by instantiating the `ChessPiece` class. However, it doesn't have to. The variable type and the object it refers to must be compatible, but their types need not be exactly the same. The relationship between a reference variable and the object it refers to is more flexible than that.

KEY CONCEPT
A polymorphic reference can refer to different types of objects over time.

The term *polymorphism* can be defined as "having many forms." A *polymorphic reference* is a reference variable that can refer to different types of objects at different points in time. The specific method invoked through a polymorphic reference (the actual code executed) can change from one invocation to the next.

Consider the following line of code:

```
obj.doIt();
```

If the reference `obj` is polymorphic, it can refer to different types of objects at different times. Thus, if that line of code is in a loop, or if it's in a method that is called more than once, that line of code could call a different version of the `doIt` method each time it is invoked.

At some point, the commitment is made to execute certain code to carry out a method invocation. This commitment is referred to as *binding* a method invocation to a method definition. In many situations, the binding of a method invocation to a method definition can occur at compile-time. For polymorphic references, however, the decision cannot be made until run-time. The method definition that is used is determined by the type of the object being referenced at the moment of invocation. This deferred commitment is called *dynamic binding* or *late binding*. It is slightly less efficient than binding at compile-time, because the decision is made during the execution of the program. This overhead is generally acceptable in light of the flexibility that a polymorphic reference provides.

KEY CONCEPT
The binding of a method invocation to its definition is performed at run-time for a polymorphic reference.

We can create a polymorphic reference in Java in two ways: using inheritance and using interfaces. Let's look at each in turn.

9.2 Polymorphism via Inheritance

When we declare a reference variable using a particular class name, it can be used to refer to any object of that class. In addition, it can refer to any object of any class that is related to its declared type by inheritance. For example, if the class `Mammal` is the parent of the class `Horse`, then a `Mammal` reference can be used to refer to an object of class `Horse`. This ability is shown in the following code segment:

```
Mammal pet;
Horse secretariat = new Horse();
pet = secretariat;  // a valid assignment
```

The ability to assign an object of one class to a reference of another class may seem like a deviation from the concept of strong typing discussed in Chapter 2, but it's not. Strong typing asserts that a variable can be assigned only a value consistent with its declared type. Well, that's what's happening here. Remember, inheritance establishes an is-a relationship. A horse *is-a* mammal. Therefore, assigning a `Horse` object to a `Mammal` reference is perfectly reasonable.

The reverse operation, assigning the `Mammal` object to a `Horse` reference, can also be done, but it requires an explicit cast. Assigning a reference in this direction is generally less useful and more likely to cause problems, because although a horse has all the functionality of a mammal, the reverse is not necessarily true.

KEY CONCEPT
A reference variable can refer to any object created from any class related to it by inheritance.

This relationship works throughout a class hierarchy. If the `Mammal` class were derived from a class called `Animal`, the following assignment would also be valid:

```
Animal creature = new Horse();
```

Carrying this idea to the limit, an `Object` reference can be used to refer to any object, because ultimately all classes are descendants of the `Object` class.

The reference variable `creature` can be used polymorphically, because at any point in time it can refer to an `Animal` object, a `Mammal` object, or a `Horse` object. Suppose that all three of these classes have a method called `move` that is implemented in different ways (because the child class overrode the definition it inherited). The following invocation calls the `move` method, but the particular version of the method it calls is determined at run-time:

```
creature.move();
```

When this line is executed, if `creature` currently refers to an `Animal` object, the `move` method of the `Animal` class is invoked. Likewise, if `creature` currently refers to a `Mammal` object, the `Mammal` version of `move` is invoked. Similarly, if it currently refers to a `Horse` object, the `Horse` version of `move` is invoked.

Of course, since `Animal` and `Mammal` represent general concepts, they may be defined as abstract classes. This situation does not eliminate the ability to have polymorphic references. Suppose the `move` method in the `Mammal` class is abstract and is given unique definitions in the `Horse`, `Dog`, and `Whale` classes (all derived from `Mammal`). A `Mammal` reference variable can be used to refer to any objects created from any of the `Horse`, `Dog`, and `Whale` classes, and it can be used to execute the `move` method on any of them, even though `Mammal` itself is abstract.

KEY CONCEPT
The type of the object, not the type of the reference, determines which version of a method is invoked.

Let's look at another situation. Consider the class hierarchy shown in Figure 9.1. The classes in it represent various types of employees that might be employed at a particular company. Let's explore an example that uses this hierarchy to pay a set of employees of various types.

The `Firm` class shown in Listing 9.1 on page 416 contains a `main` driver that creates a `Staff` of employees and invokes the `payday` method to pay them all. The program output includes information about each employee and how much each is paid (if anything).

The `Staff` class shown in Listing 9.2 on page 417 maintains an array of objects that represent individual employees of various kinds. Note that the array is declared to hold `StaffMember` references, but it is actually filled with objects created from several other classes, such as `Executive` and `Employee`. These classes are all descendants of the `StaffMember` class, so the assignments are valid. The `staffList` array is filled with polymorphic references.

The `payday` method of the `Staff` class scans through the list of employees, printing their information and invoking their `pay` methods to determine how much each employee should be paid. The invocation of the `pay` method is polymorphic because each class has its own version of the `pay` method.

The `StaffMember` class shown in Listing 9.3 on page 419 is abstract. It does not represent a particular type of employee and is not intended to be instantiated. Rather, it serves as the ancestor of all employee classes and contains information that applies to all employees. Each employee has a name, address, and phone number, so variables to store these values are declared in the `StaffMember` class and are inherited by all descendants.

The `StaffMember` class contains a `toString` method to return the information managed by the `StaffMember` class. It also contains an abstract method called `pay`, which takes no parameters and returns a value of type `double`. At the generic `StaffMember` level, it would be inappropriate to give a definition for this method. However, each descendant of `StaffMember` provides its own specific definition for `pay`.

This example shows the essence of polymorphism. Each class knows best how it should handle a specific behavior, in this case paying an employee. Yet in one sense it's all the same behavior—the employee is getting paid. Polymorphism lets us treat similar objects in consistent but unique ways.

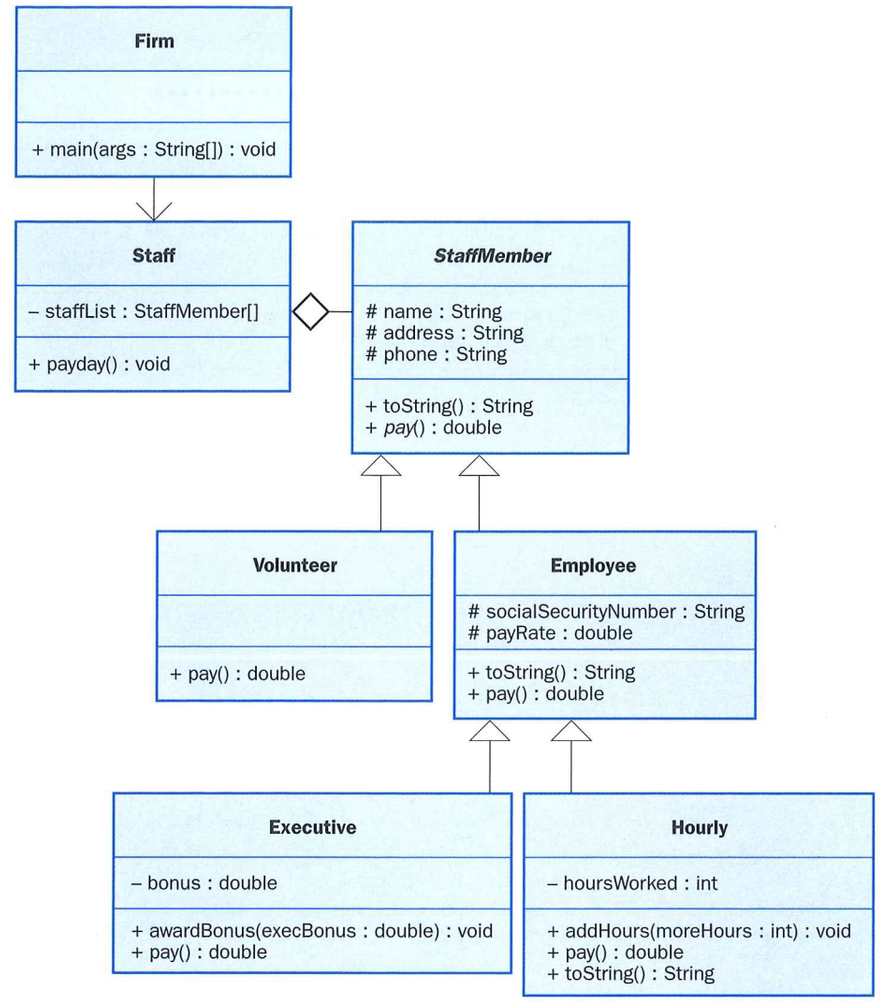

FIGURE 9.1 A class hierarchy of employees

Because `pay` is defined abstractly in `StaffMember`, the `payday` method of `Staff` can pay each employee polymorphically. If the `pay` method were not established in `StaffMember`, the compiler would complain when `pay` was invoked through an element of the `staffList` array. The abstract method guarantees the compiler that any object referenced through the `staffList` array has a `pay` method defined for it.

LISTING 9.1

```
//********************************************************************
//  Firm.java       Java Foundations
//
//  Demonstrates polymorphism via inheritance.
//********************************************************************

public class Firm
{
   //-----------------------------------------------------------------
   //  Creates a staff of employees for a firm and pays them.
   //-----------------------------------------------------------------
   public static void main(String[] args)
   {
      Staff personnel = new Staff();

      personnel.payday();
   }
}
```

OUTPUT

```
Name: Tony
Address: 123 Main Line
Phone: 555-0469
Social Security Number: 123-45-6789
Paid: 2923.07
-----------------------------------
Name: Paulie
Address: 456 Off Line
Phone: 555-0101
Social Security Number: 987-65-4321
Paid: 1246.15
-----------------------------------
Name: Vito
Address: 789 Off Rocker
Phone: 555-0000
Social Security Number: 010-20-3040
Paid: 1169.23
-----------------------------------
```


LISTING 9.1 *continued*

```
Name: Michael
Address: 678 Fifth Ave.
Phone: 555-0690
Social Security Number: 958-47-3625
Current hours: 40
Paid: 422.0
-----------------------------------
Name: Adrianna
Address: 987 Babe Blvd.
Phone: 555-8374
Thanks!
-----------------------------------
Name: Benny
Address: 321 Dud Lane
Phone: 555-7282
Thanks!
-----------------------------------
```

LISTING 9.2

```
//********************************************************************
//  Staff.java      Java Foundations
//
//  Represents the personnel staff of a particular business.
//********************************************************************

public class Staff
{
   private StaffMember[] staffList;

   //-----------------------------------------------------------------
   //  Constructor: Sets up the list of staff members.
   //-----------------------------------------------------------------
   public Staff()
   {
      staffList = new StaffMember[6];
```

LISTING 9.2 *continued*

```
      staffList[0] = new Executive("Tony", "123 Main Line",
         "555-0469", "123-45-6789", 2423.07);
      staffList[1] = new Employee("Paulie", "456 Off Line",
         "555-0101", "987-65-4321", 1246.15);
      staffList[2] = new Employee("Vito", "789 Off Rocker",
         "555-0000", "010-20-3040", 1169.23);

      staffList[3] = new Hourly("Michael", "678 Fifth Ave.",
         "555-0690", "958-47-3625", 10.55);

      staffList[4] = new Volunteer("Adrianna", "987 Babe Blvd.",
         "555-8374");
      staffList[5] = new Volunteer("Benny", "321 Dud Lane",
         "555-7282");

      ((Executive)staffList[0]).awardBonus(500.00);

      ((Hourly)staffList[3]).addHours(40);
   }

   //-----------------------------------------------------------------
   //  Pays all staff members.
   //-----------------------------------------------------------------
   public void payday()
   {
      double amount;

      for (int count=0; count < staffList.length; count++)
      {
         System.out.println(staffList[count]);

         amount = staffList[count].pay();    // polymorphic

         if (amount == 0.0)
            System.out.println("Thanks!");
         else
            System.out.println("Paid: " + amount);

         System.out.println("-----------------------------------");
      }
   }
}
```


LISTING 9.3

```
//********************************************************************
//  StaffMember.java       Java Foundations
//
//  Represents a generic staff member.
//********************************************************************

abstract public class StaffMember
{
   protected String name;
   protected String address;
   protected String phone;

   //-----------------------------------------------------------------
   //  Constructor: Sets up this staff member using the specified
   //  information.
   //-----------------------------------------------------------------
   public StaffMember(String eName, String eAddress, String ePhone)
   {
      name = eName;
      address = eAddress;
      phone = ePhone;
   }

   //-----------------------------------------------------------------
   //  Returns a string including the basic employee information.
   //-----------------------------------------------------------------
   public String toString()
   {
      String result = "Name: " + name + "\n";

      result += "Address: " + address + "\n";
      result += "Phone: " + phone;

     return result;
   }

   //-----------------------------------------------------------------
   //  Derived classes must define the pay method for each type of
   //  employee.
   //-----------------------------------------------------------------
   public abstract double pay();
}
```

LISTING 9.4

```
//********************************************************************
//  Volunteer.java       Java Foundations
//
//  Represents a staff member that works as a volunteer.
//********************************************************************

public class Volunteer extends StaffMember
{
   //-----------------------------------------------------------------
   //  Constructor: Sets up this volunteer using the specified
   //  information.
   //-----------------------------------------------------------------
   public Volunteer(String eName, String eAddress, String ePhone)
   {
      super(eName, eAddress, ePhone);
   }

   //-----------------------------------------------------------------
   //  Returns a zero pay value for this volunteer.
   /-----------------------------------------------------------------
   public double pay()
   {
      return 0.0;
   }
}
```

The `Volunteer` class shown in Listing 9.4 represents a person who is not compensated monetarily for his or her work. We keep track only of a volunteer's basic information, which is passed into the constructor of `Volunteer`, which in turn passes it to the `StaffMember` constructor using the `super` reference. The `pay` method of `Volunteer` simply returns a zero pay value. If `pay` had not been overridden, the `Volunteer` class would have been considered abstract and could not have been instantiated.

VideoNote
Exploring the `Firm` program

Note that when a volunteer gets "paid" in the `payday` method of `Staff`, a simple expression of thanks is printed. In all other situations, where the pay value is greater than zero, the payment itself is printed.

The `Employee` class shown in Listing 9.5 represents an employee who gets paid at a particular rate each pay period. The pay rate and the employee's Social Security number are passed, along with the other basic information, to the `Employee` constructor. The basic information is passed to the constructor of `StaffMember` using the `super` reference.

The `toString` method of `Employee` is overridden to concatenate the additional information that `Employee` manages to the information returned by the parent's version of `toString`, which is called using the `super` reference. The `pay` method of an `Employee` simply returns the pay rate for that employee.

LISTING 9.5

```
//********************************************************************
//  Employee.java       Java Foundations
//
//  Represents a general paid employee.
//********************************************************************

public class Employee extends StaffMember
{
   protected String socialSecurityNumber;
   protected double payRate;

   //-----------------------------------------------------------------
   //  Constructor: Sets up this employee with the specified
   //  information.
   //-----------------------------------------------------------------
   public Employee(String eName, String eAddress, String ePhone,
                   String socSecNumber, double rate)
   {
      super(eName, eAddress, ePhone);

      socialSecurityNumber = socSecNumber;
      payRate = rate;
   }

   //-----------------------------------------------------------------
   //  Returns information about an employee as a string.
   //-----------------------------------------------------------------
   public String toString()
   {
      String result = super.toString();
```

LISTING 9.5 *continued*

```
      result += "\nSocial Security Number: " + socialSecurityNumber;

      return result;
   }

   //-----------------------------------------------------------------
   //  Returns the pay rate for this employee.
   //-----------------------------------------------------------------
   public double pay()
   {
      return payRate;
   }
}
```

The `Executive` class shown in Listing 9.6 represents an employee who may earn a bonus in addition to his or her normal pay rate. The `Executive` class is derived from `Employee` and therefore inherits from both `StaffMember` and `Employee`. The constructor of `Executive` passes along its information to the `Employee` constructor and sets the executive bonus to zero.

A bonus is awarded to an executive using the `awardBonus` method. This method is called in the `payday` method in `Staff` for the only executive that is part of the `staffList` array. Note that the generic `StaffMember` reference must be cast into an `Executive` reference to invoke the `awardBonus` method (because it doesn't exist for a `StaffMember`).

The `Executive` class overrides the `pay` method so that it first determines the payment as it would for any employee; then it adds the bonus. The `pay` method of the `Employee` class is invoked using `super` to obtain the normal payment amount. This technique is better than using just the `payRate` variable, because if we choose to change how `Employee` objects get paid, the change will automatically be reflected in `Executive`. After the bonus is awarded, it is reset to zero.

The `Hourly` class shown in Listing 9.7 on page 424 represents an employee whose pay rate is applied on an hourly basis. It keeps track of the number of hours worked in the current pay period, which can be modified by calls to the `addHours` method. This method is called from the `payday` method of `Staff`. The `pay` method of `Hourly` determines the payment on the basis of the number of hours worked and then resets the hours to zero.

LISTING 9.6

```
//********************************************************************
//  Executive.java       Java Foundations
//
//  Represents an executive staff member, who can earn a bonus.
//********************************************************************

public class Executive extends Employee
{
   private double bonus;

   //-----------------------------------------------------------------
   //  Constructor: Sets up this executive with the specified
   //  information.
   //-----------------------------------------------------------------
   public Executive(String eName, String eAddress, String ePhone,
                    String socSecNumber, double rate)
   {
      super(eName, eAddress, ePhone, socSecNumber, rate);

      bonus = 0; // bonus has yet to be awarded
   }

   //-----------------------------------------------------------------
   //  Awards the specified bonus to this executive.
   //-----------------------------------------------------------------
   public void awardBonus(double execBonus)
   {
      bonus = execBonus;
   }

   //-----------------------------------------------------------------
   //  Computes and returns the pay for an executive, which is the
   //  regular employee payment plus a one-time bonus.
   //-----------------------------------------------------------------
   public double pay()
   {
      double payment = super.pay() + bonus;

      bonus = 0;

      return payment;
   }
}
```

LISTING 9.7

```
//********************************************************************
//  Hourly.java       Java Foundations
//
//  Represents an employee that gets paid by the hour.
//********************************************************************

public class Hourly extends Employee
{
   private int hoursWorked;

   //-----------------------------------------------------------------
   //  Constructor: Sets up this hourly employee using the specified
   //  information.
   //-----------------------------------------------------------------
   public Hourly(String eName, String eAddress, String ePhone,
                 String socSecNumber, double rate)
   {
      super(eName, eAddress, ePhone, socSecNumber, rate);

      hoursWorked = 0;
   }

   //-----------------------------------------------------------------
   //  Adds the specified number of hours to this employee's
   //  accumulated hours.
   //-----------------------------------------------------------------
   public void addHours(int moreHours)
   {
      hoursWorked += moreHours;
   }

   //-----------------------------------------------------------------
   //  Computes and returns the pay for this hourly employee.
   //-----------------------------------------------------------------
   public double pay()
   {
      double payment = payRate * hoursWorked;

      hoursWorked = 0;

      return payment;
   }
```


LISTING 9.7 *continued*

```
   //-----------------------------------------------------------------
   //  Returns information about this hourly employee as a string.
   //-----------------------------------------------------------------
   public String toString()
   {
      String result = super.toString();

      result += "\nCurrent hours: " + hoursWorked;
      count--;

      return result;
   }
}
```

9.3 Interfaces

In Chapter 5 we used the term *interface* to refer to the set of public methods through which we can interact with an object. That definition is consistent with our use of it in this section, but now we are going to formalize this concept using a Java language construct. Interfaces provide another way to create polymorphic references.

A Java *interface* is a collection of constants and abstract methods. As discussed in Chapter 8, an abstract method is a method that does not have an implementation. That is, there is no body of code defined for an abstract method. The header of the method, including its parameter list, is simply followed by a semicolon. An interface cannot be instantiated.

KEY CONCEPT
An interface is a collection of abstract methods and therefore cannot be instantiated.

Listing 9.8 on page 426 shows an interface called `Encryptable`. It contains two abstract methods: `encrypt` and `decrypt`.

An abstract method can be preceded by the reserved word `abstract`, although in interfaces it usually is not. Methods in interfaces have public visibility by default.

A class *implements* an interface by providing method implementations for each of the abstract methods defined in the interface. The `Secret` class, shown in Listing 9.9 on page 426, implements the `Encryptable` interface.

LISTING 9.8

```
//********************************************************************
//  Encryptable.java       Java Foundations
//
//  Represents the interface for an object that can be encrypted
//  and decrypted.
//********************************************************************

public interface Encryptable
{
   public void encrypt();
   public String decrypt();
}
```

LISTING 9.9

```
//********************************************************************
//  Secret.java       Java Foundations
//
//  Represents a secret message that can be encrypted and decrypted.
//********************************************************************

import java.util.Random;

public class Secret implements Encryptable
{
   private String message;
   private boolean encrypted;
   private int shift;
   private Random generator;

   //-----------------------------------------------------------------
   //  Constructor: Stores the original message and establishes
   //  a value for the encryption shift.
   //-----------------------------------------------------------------
   public Secret(String msg)
   {
      message = msg;
      encrypted = false;
```


LISTING 9.9 *continued*

```
      generator = new Random();
      shift = generator.nextInt(10) + 5;
   }

   //-----------------------------------------------------------------
   //  Encrypts this secret using a Caesar cipher. Has no effect if
   //  this secret is already encrypted.
   //-----------------------------------------------------------------
   public void encrypt()
   {
      if (!encrypted)
      {
         String masked = "";
         for (int index=0; index < message.length(); index++)
            masked = masked + (char)(message.charAt(index)+shift);
         message = masked;
         encrypted = true;
      }
   }

   //-----------------------------------------------------------------
   //  Decrypts and returns this secret. Has no effect if this
   //  secret is not currently encrypted.
   //-----------------------------------------------------------------
   public String decrypt()
   {
      if (encrypted)
      {
         String unmasked = "";
         for (int index=0; index < message.length(); index++)
            unmasked = unmasked + (char)(message.charAt(index)-shift);
         message = unmasked;
         encrypted = false;
      }

      return message;
   }

   //-----------------------------------------------------------------
   //  Returns true if this secret is currently encrypted.
   //-----------------------------------------------------------------
   public boolean isEncrypted()
```

LISTING 9.9 *continued*

```
    {
       return encrypted;
    }

    //-----------------------------------------------------------------
    //  Returns this secret (may be encrypted).
    //-----------------------------------------------------------------
    public String toString()
    {
       return message;
    }
}
```

A class that implements an interface uses the reserved word `implements` followed by the interface name in the class header. If a class asserts that it implements a particular interface, it must provide a definition for all methods in the interface. The compiler will produce errors if any of the methods in the interface is not given a definition in the class.

In the class `Secret`, both the `encrypt` method and the `decrypt` method are implemented, which satisfies the contract established by the interface. These methods must be declared with the same signatures as their abstract counterparts in the interface. In the `Secret` class, the encryption is implemented using a simple Caesar cipher, which shifts the characters of the message a certain number of places. Another class that implements the `Encryptable` interface may use a completely different technique for encryption.

Note that the `Secret` class also implements additional methods that are not part of the `Encryptable` interface. Specifically, it defines the methods `isEncrypted` and `toString`, which have nothing to do with the interface. The interface guarantees that the class implements certain methods, but it does not restrict it from having others. In fact, it is common for a class that implements an interface to have other methods.

Listing 9.10 shows a program called `SecretTest`, which creates some `Secret` objects.

An interface and its relationship to a class can be shown in a UML class diagram. An interface is represented similarly to a class node except that the designation

LISTING 9.10

```
//********************************************************************
//  SecretTest.java       Java Foundations
//
//  Demonstrates the use of a formal interface.
//********************************************************************

public class SecretTest
{
   //-----------------------------------------------------------------
   //  Creates a Secret object and exercises its encryption.
   //-----------------------------------------------------------------
   public static void main(String[] args)
   {
      Secret hush = new Secret("Wil Wheaton is my hero!");
      System.out.println(hush);

      hush.encrypt();
      System.out.println(hush);

      hush.decrypt();
      System.out.println(hush);
   }
}
```

OUTPUT

```
Wil Wheaton is my hero!
asv*arok~yx*s}*w?*ro|y+
Wil Wheaton is my hero!
```

`<<interface>>` is inserted above the interface name. A dotted arrow with a triangular arrowhead is drawn from the class to the interface that it implements. Figure 9.2 on page 430 shows a UML class diagram for the `SecretTest` program.

Multiple classes can implement the same interface, providing their own definitions for the methods. For example, we could implement a class called `Password` that also implements the `Encryptable` interface. And, as mentioned earlier, each class that implements an interface may do so in different ways. The interface specifies which methods are implemented, not how they are implemented.

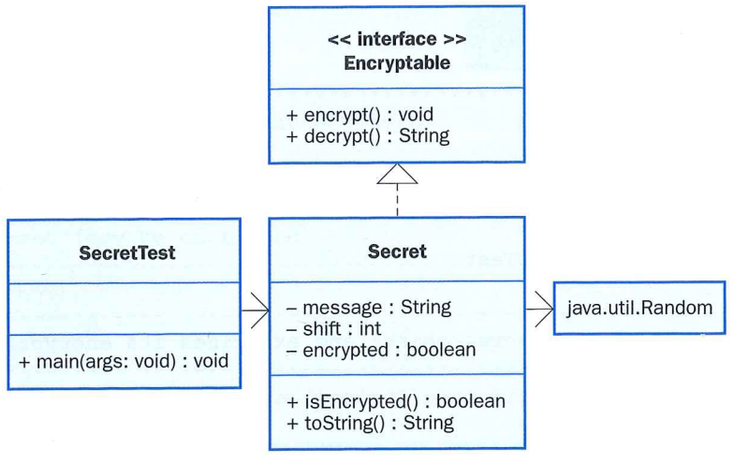

FIGURE 9.2 A UML class diagram for the `SecretTest` program

A class can implement more than one interface. In these cases, the class must provide an implementation for all methods in all interfaces listed. To show that a class implements multiple interfaces, they are listed in the `implements` clause, separated by commas. Here is an example:

```
class ManyThings implements Interface1, Interface2, Interface3
{

 // implements all methods of all interfaces

}
```

In addition to, or instead of, abstract methods, an interface can contain constants defined using the `final` modifier. When a class implements an interface, it gains access to all the constants defined in it.

Interface Hierarchies

The concept of inheritance can be applied to interfaces as well as to classes. That is, one interface can be derived from another interface. These relationships can form an *interface hierarchy*, which is similar to a class hierarchy. Inheritance relationships between interfaces are shown in UML diagrams using the same connection (an arrow with an open arrowhead) that is used to show inheritance relationships between classes.

KEY CONCEPT
Inheritance can be applied to interfaces so that one interface can be derived from another.

When a parent interface is used to derive a child interface, the child inherits all abstract methods and constants of the parent. Any class that implements the child interface must implement all of the methods. There are no visibility issues when dealing with inheritance between interfaces (as there are with protected and private members of a class), because all members of an interface are public.

Class hierarchies and interface hierarchies do not overlap. That is, an interface cannot be used to derive a class, and a class cannot be used to derive an interface. A class and an interface interact only when a class is designed to implement a particular interface.

Before we see how interfaces support polymorphism, let's take a look at a couple of useful interfaces that are defined in the Java standard class library: `Comparable` and `Iterator`.

The `Comparable` Interface

The Java standard class library contains interfaces as well as classes. The `Comparable` interface, for example, is defined in the `java.lang` package. The `Comparable` interface contains only one method, `compareTo`, which takes an object as a parameter and returns an integer.

The purpose of this interface is to provide a common mechanism for comparing one object to another. One object calls the method and passes another as a parameter as follows:

```
if (obj1.compareTo(obj2) < 0)
    System.out.println("obj1 is less than obj2");
```

As specified by the documentation for the interface, the integer that is returned from the `compareTo` method should be negative if `obj1` is less than `obj2`, 0 if they are equal, and positive if `obj1` is greater than `obj2`. It is up to the designer of each class to decide what it means for one object of that class to be less than, equal to, or greater than another.

In Chapter 4, we mentioned that the `String` class contains a `compareTo` method that operates in this manner. Now we can clarify that the `String` class has this method because it implements the `Comparable` interface. The `String` class implementation of this method bases the comparison of strings on the lexicographic ordering defined by the Unicode character set.

The `Iterator` Interface

The `Iterator` interface is another interface defined in the Java standard class library. It is used by a class that represents a collection of objects, providing a means to move through the collection one object at a time.

In Chapter 4, we defined the concept of an iterator, using a loop to process all elements in the collection. Most iterators, including objects of the `Scanner` class, are defined using the `Iterator` interface.

The two primary methods in the `Iterator` interface are `hasNext`, which returns a boolean result, and `next`, which returns an object. Neither of these methods takes any parameters. The `hasNext` method returns true if there are items left to process, and `next` returns the next object. It is up to the designer of the class that implements the `Iterator` interface to decide in what order objects will be delivered by the `next` method.

We should note that, in accordance with the spirit of the interface, the `next` method does not remove the object from the underlying collection; it simply returns a reference to it. The `Iterator` interface also has a method called `remove`, which takes no parameters and has a `void` return type. A call to the `remove` method removes the object that was most recently returned by the `next` method from the underlying collection.

We've seen how we can use iterators to process information from the `Scanner` class (in Chapter 4) and from arrays (in Chapter 7). Recall that the foreach version of the `for` loop simplifies this processing in many cases. We will continue to use `iterators` as appropriate. They are an important part of the development of collection classes, which we discuss in detail in the later chapters of this text (Chapters 14 and beyond).

9.4 Polymorphism via Interfaces

Now let's examine how we can create polymorphic references using interfaces. As we've seen many times, a class name can be used to declare the type of an object reference variable. Similarly, an interface name can be used as the type of a reference variable as well. An interface reference variable can be used to refer to any object of any class that implements that interface.

KEY CONCEPT
An interface name can be used to declare an object reference variable.

Suppose we declare an interface called `Speaker` as follows:

```
public interface Speaker
{
   public void speak();
   public void announce(String str);
}
```

The interface name, `Speaker`, can now be used to declare an object reference variable:

```
Speaker current;
```


The reference variable `current` can be used to refer to any object of any class that implements the `Speaker` interface. For example, if we define a class called `Philosopher` such that it implements the `Speaker` interface, we can then assign a `Philosopher` object to a `Speaker` reference as follows:

```
current = new Philosopher();
```

This assignment is valid because a `Philosopher` is a `Speaker`. In this sense, the relationship between a class and its interface is the same as the relationship between a child class and its parent. It is an is-a relationship, similar to the relationship created via inheritance. And that relationship forms the basis of the polymorphism.

KEY CONCEPT
An interface reference can refer to any object of any class that implements that interface.

The flexibility of an interface reference allows us to create polymorphic references. As we saw earlier in this chapter, using inheritance, we can create a polymorphic reference that can refer to any one of a set of objects as long as they are related by inheritance. Using interfaces, we can create similar polymorphic references among objects that implement the same interface.

For example, if we create a class called `Dog` that also implements the `Speaker` interface, it can be assigned to a `Speaker` reference variable as well. The same reference variable, in fact, can at one point refer to a `Philosopher` object and then later refer to a `Dog` object. The following lines of code illustrate this:

```
Speaker guest;
guest = new Philosopher();
guest.speak();
guest = new Dog();
guest.speak();
```

In this code, the first time the `speak` method is called, it invokes the `speak` method defined in the `Philosopher` class. The second time it is called, it invokes the `speak` method of the `Dog` class. As with polymorphic references via inheritance, it is not the type of the reference that determines which method gets invoked; this is based on the type of the object that the reference points to at the moment of invocation.

Note that when we are using an interface reference variable, we can invoke only the methods defined in the interface, even if the object it refers to has other methods to which it can respond. For example, suppose the `Philosopher` class also defined a public method called `pontificate`. The second line of the following code segment would generate a compiler error, even though the object can in fact contain the `pontificate` method:

```
Speaker special = new Philosopher();
special.pontificate();      // generates a compiler error
```

The problem is that the compiler can determine only that the object is a `Speaker`, and therefore can guarantee only that the object can respond to the `speak` and `announce` methods. Because the reference variable `special` could refer to a `Dog` object (which cannot pontificate), it does not allow the invocation. If we know in a particular situation that such an invocation is valid, we can cast the object into the appropriate reference so that the compiler will accept it, as follows:

```
((Philosopher)special).pontificate();
```

KEY CONCEPT

A parameter to a method can be polymorphic, which gives the method flexible control of its arguments.

Just as with polymorphic references based in inheritance, we can use an interface name as the type of a method parameter. In such situations, any object of any class that implements the interface can be passed into the method. For example, the following method takes a `Speaker` object as a parameter. Therefore, both a `Dog` object and a `Philosopher` object can be passed into it in separate invocations:

```
public void sayIt(Speaker current)
{
   current.speak();
}
```

Using a polymorphic reference as the formal parameter to a method is a powerful technique. It allows the method to control the types of parameters passed into it, yet gives it the flexibility to accept arguments of various types.

Let's now examine a particular use of polymorphism via interfaces.

Event Processing

In Chapter 6 we examined the processing of events in a Java GUI. Recall that, in order to respond to an event, we must establish a relationship between an event listener object and a particular component that may fire the event. We establish the relationship between the listener and the component it listens to by making a method call that adds the listener to the component. This situation is actually an example of polymorphism.

Suppose a class called `MyButtonListener` represents an action listener. To set up a listener to respond to a `JButton` object, we might do the following:

```
JButton button = new JButton();
button.addActionListener(new MyButtonListener());
```

Once this relationship is established, the listener will respond whenever the button fires an action event (because the user pressed it). Now think about the `addActionListener` method carefully. It is a method of the `JButton` class,

which was written by someone at Sun Microsystems years ago. On the other hand, we might have written the `MyButtonListener` class today. How can a method written years ago take a parameter whose class was just written?

KEY CONCEPT

The relationship between a listener and the component it listens to is established using polymorphism.

The answer is polymorphism. If you examine the source code for the `addActionListener` method, you'll discover that it accepts a parameter of type `ActionListener`, the interface. Therefore, instead of accepting a parameter of only one object type, the `addActionListener` method can accept any object of any class that implements the `ActionListener` interface. All other methods that add listeners work in similar ways.

The `JButton` object doesn't know anything in particular about the object that is passed to the `addActionListener` method, except for the fact that it implements the `ActionListener` interface (otherwise, the code wouldn't compile). The `JButton` object simply stores the listener object and invokes its `performAction` method when the event occurs.

In Chapter 6 we mentioned that we can also create a listener by extending an adaptor class. This is another example of polymorphism via interfaces, even though the listener class is created via inheritance. Each adaptor class is written to implement the appropriate listener interface, providing empty methods for all event handlers. By extending an adaptor class, the new listener class automatically implements the corresponding listener interface. And that is what really makes it a listener such that it can be passed to an appropriate add listener method.

Thus, no matter how a listener object is created, we are using polymorphism via interfaces to set up the relationship between a listener and the component it listens to. GUI events are a wonderful example of the power and versatility provided by polymorphism.

Polymorphism, whether implemented via inheritance or interfaces, is a fundamental object-oriented technique that we will use as appropriate throughout the remainder of this text.

Summary of Key Concepts

- A polymorphic reference can refer to different types of objects over time.
- The binding of a method invocation to its definition is performed at run-time for a polymorphic reference.
- A reference variable can refer to any object created from any class related to it by inheritance.
- The type of the object, not the type of the reference, determines which version of a method is invoked.
- An interface is a collection of abstract methods and therefore cannot be instantiated.
- Inheritance can be applied to interfaces so that one interface can be derived from another.
- An interface name can be used to declare an object reference variable.
- An interface reference can refer to any object of any class that implements that interface.
- A parameter to a method can be polymorphic, which gives the method flexible control of its arguments.
- The relationship between a listener and the component it listens to is established using polymorphism.

Summary of Terms

binding The process of determining which method definition is used to fulfill a given method invocation.

dynamic binding The binding of a method invocation to its definition at run-time. Also called late binding.

interface A collection of abstract methods, used to define a set of operations that can be used to interact with an object.

interface hierarchy The hierarchy formed when interfaces are derived from other interfaces. Interface hierarchies are distinct from class hierarchies.

polymorphism The ability to define an operation that has more than one meaning by having the operation dynamically bound to methods of various objects.

polymorphic reference A reference variable that can refer to different types of objects at different points in time.

Self-Review Questions

SR 9.1 What is polymorphism?

SR 9.2 How does inheritance support polymorphism?

SR 9.3 How is overriding related to polymorphism?

SR 9.4 Why is the `StaffMember` class in the `Firm` example declared as abstract?

SR 9.5 Why is the `pay` method declared in the `StaffMember` class, given that it is abstract and has no body at that level?

SR 9.6 What is the difference between a class and an interface?

SR 9.7 How do class hierarchies and interface hierarchies intersect?

SR 9.8 Describe the `Comparable` interface.

SR 9.9 How can polymorphism be accomplished using interfaces?

Exercises

EX 9.1 Draw and annotate a class hierarchy that represents various types of faculty at a university. Show what characteristics would be represented in the various classes of the hierarchy. Explain how polymorphism could play a role in the process of assigning courses to each faculty member.

EX 9.2 Draw and annotate a class hierarchy that represents various types of animals in a zoo. Show what characteristics would be represented in the various classes of the hierarchy. Explain how polymorphism could play a role in guiding the feeding of the animals.

EX 9.3 Draw and annotate a class hierarchy that represents various types of sales transactions in a store (cash, credit, and so on). Show what characteristics would be represented in the various classes of the hierarchy. Explain how polymorphism could play a role in the payment process.

EX 9.4 What would happen if the `pay` method were not defined as an abstract method in the `StaffMember` class of the `Firm` program?

EX 9.5 Create an interface called `Visible` that includes two methods: `makeVisible` and `makeInvisible`. Both methods should take no parameters and should return a boolean result. Describe how a class might implement this interface.

EX 9.6 Draw a UML class diagram that shows the relationships among the elements of Exercise 9.5.

EX 9.7 Create an interface called `VCR` that has methods that represent the standard operations on a video cassette recorder (play, stop, and so on). Define the method signatures any way you desire. Describe how a class might implement this interface.

EX 9.8 Draw a UML class diagram that shows the relationships among the elements of Exercise 9.7.

EX 9.9 Explain how a call to the `addMouseListener` method in a GUI-based program represents a polymorphic situation.

Programming Projects

PP 9.1 Modify the `Firm` example from this chapter such that it accomplishes its polymorphism using an interface called `Payable`.

PP 9.2 Modify the `Firm` example from this chapter such that all employees can be given different vacation options depending on their classification. Modify the driver program to demonstrate this new functionality.

PP 9.3 Modify the `RationalNumber` class from Chapter 5 so that it implements the `Comparable` interface. To perform the comparison, compute an equivalent floating point value from the numerator and denominator for both `RationalNumber` objects, and then compare them using a tolerance value of 0.0001. Write a `main` driver to test your modifications.

PP 9.4 Create a class called `Password` that implements the `Encryptable` interface from this chapter. Then create a `main` driver that instantiates a `Secret` object and a `Password` object, using the same reference variable, and exercises their methods. Use any type of encryption desired for `Password`, other than the Caesar cipher used by `Secret`.

PP 9.5 Implement the `Speaker` interface defined in this chapter. Create three classes that implement `Speaker` in various ways. Create a driver class whose `main` method instantiates some of these objects and tests their abilities.

PP 9.6 Design a Java interface called `Priority` that includes two methods: `setPriority` and `getPriority`. The interface should define a way to establish numeric priority among a set of objects. Design and implement a class called `Task` that represents a task (such as on a to-do list) that implements the `Priority` interface. Create a driver class to exercise some `Task` objects.

PP 9.7 Modify the `Task` class from Programming Project 9.6 so that it also implements the `Comparable` interface from the Java standard class library. Implement the interface such that the tasks are ranked by priority. Create a driver class whose `main` method shows these new features of `Task` objects.

PP 9.8 Design a Java interface called `Lockable` that includes the following methods: `setKey`, `lock`, `unlock`, and `locked`. The `setKey`, `lock`, and `unlock` methods take an integer parameter that represents the key. The `setKey` method establishes the key. The `lock` and `unlock` methods lock and unlock the object, but only if the key passed in is correct. The `locked` method returns a boolean that indicates whether or not the object is locked. A `Lockable` object represents an object whose regular methods are protected: If the object is locked, the methods cannot be invoked; if it is unlocked, they can be invoked. Redesign and implement a version of the `Coin` class from Chapter 5 so that it is `Lockable`.

PP 9.9 Redesign and implement a version of the `Account` class from Chapter 5 so that it is `Lockable` as defined by Programming Project 9.8.

Answers to Self-Review Questions

SRA 9.1 Polymorphism is the ability of a reference variable to refer to objects of various types at different times. A method invoked through such a reference is bound to different method definitions at different times, depending on the type of the object referenced.

SRA 9.2 In Java, a reference variable declared using a parent class can be used to refer to an object of the child class. If both classes contain a method with the same signature, the parent reference can be polymorphic.

SRA 9.3 When a child class overrides the definition of a parent's method, two versions of that method exist. If a polymorphic reference is used to invoke the method, the version of the method that is invoked is determined by the type of the object being referred to, not by the type of the reference variable.

SRA 9.4 The `StaffMember` class is abstract because it is not intended to be instantiated. It serves as a placeholder in the inheritance hierarchy to help organize and manage the objects polymorphically.

SRA 9.5 The `pay` method has no meaning at the `StaffMember` level, so it is declared as abstract. But by declaring it there, we guarantee that every object of its children will have a `pay` method. This allows us to create an array of `StaffMember` objects, which is actually filled with various types of staff members, and to pay each one. The details of being paid are determined by each class, as appropriate.

SRA 9.6 A class can be instantiated; an interface cannot. An interface can contain only abstract methods and constants. A class provides the implementation for an interface.

SRA 9.7 Class hierarchies and interface hierarchies do not intersect. A class can be used to derive a new class, and an interface can be used to derive a new interface, but these two types of hierarchies do not overlap.

SRA 9.8 The `Comparable` interface contains a single method called `compareTo`, which should return an integer that is less than zero, equal to zero, or greater than zero if the executing object is less than, equal to, or greater than the object to which it is being compared, respectively.

SRA 9.9 An interface name can be used as the type of a reference. Such a reference variable can refer to any object of any class that implements that interface. Because all classes implement the same interface, they have methods with common signatures, which can be dynamically bound.

10 Exceptions

CHAPTER OBJECTIVES

- Discuss the purpose of exceptions.
- Examine exception messages and the call stack trace.
- Examine the `try-catch` statement for handling exceptions.
- Explore the concept of exception propagation.
- Describe the exception class hierarchy in the Java standard class library.
- Explore I/O exceptions and the ability to write text files.

Exception handling is an important part of an object-oriented software system. Exceptions represent problems or unusual situations that may occur in a program. Java provides various ways to handle exceptions when they occur. We explore the class hierarchy from the Java standard library used to define exceptions, as well as the ability to define our own exception objects. This chapter also discusses the use of exceptions when dealing with input and output, and it presents an example that writes a text file.