Chapter 12

Polymorphism

The term *polymorphic* has Greek roots and means roughly "many forms." (*poly* = many, *morphos* = form. Morphos is related to the Greek god Morphus, who could appear to sleeping individuals in any form he wished and hence was truly polymorphic.) In biology, a polymorphic species is one, such as *Homo Sapiens*, that is characterized by the occurrence of different forms or color types in individual organisms or among organisms. In chemistry, a polymorphic compound is one that can crystallize in at least two distinct forms, such as carbon, which can crystallize both as graphite and as diamond.

12.1 Varieties of Polymorphism

In object-oriented languages, polymorphism is a natural result of the *is-a* relationship and of the mechanisms of message passing, inheritance, and the concept of substitutability. One of the great strengths of the OOP approach is that these devices can be combined in a variety of ways, yielding a number of techniques for code sharing and reuse.

Pure polymorphism occurs when a single function can be applied to arguments of a variety of types. In pure polymorphism, there is one function (code body) and a number of interpretations. The other extreme occurs when we have a number of different functions (code bodies) all denoted by the same name–a situation known as *overloading* or sometimes *ad hoc polymorphism.* Between these two extremes are *overriding* and *deferred methods.*[1]

[1]Note that there is little agreement regarding terminology in the programming language community. In [Horowitz 1984], [Marcotty 1987], [MacLennan 1987], and [Pinson 1988] for example, *polymorphism* is defined in a manner roughly equivalent to what we are here calling *overloading.* In [Sethi 1989] and [Meyer 1988a] and in the functional programming languages community (such as [Wikström 1987, Milner 1990]), the term is reserved for what we are calling *pure polymorphism.* Other authors use the term for one, two, or all of the mechanisms described in this chapter. Two complete, but technically daunting, analyses are [Cardelli 1985] and [Danforth 1988].

12.2 Polymorphic Variables

With the exception of overloading, polymorphism in object-oriented languages is made possible only by the existence of *polymorphic variables* and the idea of substitutability. A polymorphic variable is one with many faces; that is, it can hold values of different types. Polymorphic variables embody the principle of substitutability. In other words, while there is an expected type for any variable the actual type can be from any value that is a subtype of the expected type.

In dynamically bound languages (such as Smalltalk), all variables are potentially polymorphic–any variable can hold values of any type. In these languages the desired type is defined by a set of expected behaviors. For example, an algorithm may make use of an array value, expecting the subscripting operations to be defined for a certain variable; any type that defines the appropriate behavior is suitable. Thus, the user could define his or her own type of array (for example, a sparse array) and, if the array operations were implemented using the same names, use this new type with an existing algorithm.

In statically typed languages, such as Java, the situation is slightly more complex. Polymorphism occurs in Java through the difference between the declared (static) class of a variable and the actual (dynamic) class of the value the variable contains.

A good example of a polymorphic variable is the array allPiles in the Solitare game presented in Chapter 9. The array was declared as maintaining a value of type CardPile, but in fact it maintains values from each of the different subclasses of the parent class. A message presented to a value from this array, such as display in the example code shown below, executes the method associated with the dynamic type of the variable and not that of the static class.

```
public class Solitaire extends Applet {
        ...
    static CardPile allPiles [ ];
        ...

    public void paint(Graphics g) {
        for (int i = 0; i < 13; i++)
            allPiles[i].display(g);
        }
        ...
}
```

12.3 Overloading

We say a function name is *overloaded* if there are two or more function bodies associated with it. Note that overloading is a necessary part of overriding, which we and will describe in the next section, but the two terms are not identical and overloading can occur without overriding.

In overloading, it is the function *name* that is polymorphic–it has many forms. Another way to think of overloading and polymorphism is that there is a single abstract function that takes various types of arguments; the actual code executed depends on the arguments given. The fact that the compiler can often determine the correct function at compile time (in a strongly typed language), and can therefore generate only a single code sequence are simply optimizations.

12.3.1 Overloading Messages in Real Life

In Chapter 1 we saw an example in which overloading occurred without overriding, when I wanted to surprise my friend with flowers for her birthday. One possible solution was to send the message sendFlowersTo to my local florist; another was to give the *same* message to my wife. Both my florist and my wife (an instance of class Spouse) would have understood the message, and both would have acted on it to produce a similar result. In a certain sense, I could have thought of sendFlowersTo as being one function understood by both my wife and my florist, but each would have used a different algorithm to respond to my request.

Note, in particular, that there was no inheritance involved in this example. The first common superclass for my wife and my florist was the category Human. But certainly the behavior sendFlowersTo was not associated with all humans. My dentist, for example, who is also a human, would not have understood the message at all.

12.3.2 Overloading and Coercion

As an example more closely tied to programming languages, suppose a programmer is developing a library of classes representing common data structures. A number of data structures can be used to maintain a collection of elements (sets, bags, dictionaries, arrays, and priority queues, for example), and these might all define a method, add, to insert a new element into the collection.

This situation–in which two totally separate functions are used to provide semantically similar actions for different data types–occurs frequently in all programming languages, not simply in object-oriented languages. Perhaps the most common example is the overloading of the addition operator, +. The code generated by a compiler for an integer addition is often radically different from the code generated for a floating-point addition, yet programmers tend to think of the operations as a single entity, the "addition" function.

In this example it is important to point out that overloading may not be the only activity taking place. A semantically separate operation, *coercion*, is also usually associated with arithmetic operations. It occurs when a value of one type is converted into one of a different type. If mixed-type arithmetic is permitted, the addition of two values may be interpreted in a number of different ways:

- There may be four different functions, corresponding to integer + integer, integer + real, real + integer, and real + real. In this case, there is overloading but no coercion.
- There may be two different functions for integer + integer and real + real. In integer + real and real + integer, the integer value is coerced by being changed into a real value. In this situation there is a combination of overloading and coercion.
- There may be only one function, for real + real addition. All arguments are coerced into being real. In this case there is coercion only, with no overloading.

12.3.3 Overloading from Separate Classes

There are two different forms of overloading that can be distinguished. One form occurs when the same function name is found in two or more classes that are not linked by inheritance. A second form occurs when two or more functions with the same name are found within one class definition. The latter form will be described in the next section.

A good example of overloading of the first type is the method isEmpty. This method is used to determine if an object is empty, however the exact meaning of empty will differ depending upon circumstances. The message is understood by the classes Vector, Hashtable and Rectangle. The first two are collection classes, and the message returns true when there are no elements in the collection. In the class Rectangle the message returns true if either the height or width of a rectangle is zero, and thus the rectangle has no area.

```
Rectangle r1 = new Rectangle ();
if (r1.isEmpty()) ...
```

Overloading Does Not Imply Similarity

There is nothing intrinsic to overloading that requires the functions associated with an overloaded name to have any semantic similarity. Consider a program that plays a card game, such as the solitaire game we examined in Chapter 9. The method draw was used to draw the image of a card on the screen. In another application we might also have included a draw method for the pack of cards, that is, to draw a single card from the top of the deck. This draw method is not even remotely similar in semantics to the draw method for the single card, and yet they share the same name.

Note that this overloading of a single name with independent and unrelated meanings should *not* necessarily be considered bad style, and generally it will not contribute to confusion. In fact, the selection of short, clear, and meaningful names such as add, draw, and so on, contributes to ease of understanding and correct use of object-oriented components. It is far simpler to remember that you can add an element to a set than to recall that to do so requires invoking the addNewElement method, or, worse, that it requires calling the routine Set_Module_Addition_Method.

All object-oriented languages permit the occurrence of methods with similar names in unrelated classes. In this case the resolution of overloaded names is determined by observation of the class of the receiver for the message. Nevertheless, this does not mean that functions or methods can be written that take arbitrary arguments. The statically typed nature of Java still requires specific declarations of all names.

12.3.4 Parameteric Overloading

Another style of overloading, in which procedures (or functions or methods) in the same context are allowed to share a name and are disambiguated by the number and type of arguments supplied, is called *parameteric overloading*; it occurs in Java as well as in some imperative languages (such as Ada) and many functional languages. Parameteric overloading is most often found in constructor functions. A new Rectangle, for example, can be created either with no arguments (generating a rectangle with size zero and northwest corner 0,0), with two integer arguments (a width and height), with four integer arguments (width, height, northwest corner), with a Point (the northwest corner, size is zero), with a Dimension (height and width, corner 0,0), or with both a Point and a Dimension.

```
Rectangle r1 = new Rectangle ();
Rectangle r2 = new Rectangle (6, 7);
Rectangle r3 = new Rectangle (10, 10, 6, 7);
Point p1 = new Point (10, 10);
Dimension d1 = new Dimension (6, 7);
Rectangle r4 = new Rectangle (p1);
Rectangle r5 = new Rectangle (d1);
Rectangle r6 = new Rectangle (p1, d1);
```

There are six different constructor functions in this class, all with the same name. The compiler decides which function to execute based on the number and type of arguments used with the function call.

Overloading is a necessary prerequisite to the other forms of polymorphism we will consider: overriding, deferred methods, and pure polymorphism. It is also often useful in reducing the "conceptual space," that is, in reducing the amount of information that the programmer must remember. Often, this reduction in programmer-memory space is just as significant as the reduction in computer-memory space permitted by code sharing.

12.4 Overriding

In Chapter 8 we described the mechanics of overriding, so it is not necessary to repeat that discussion here. Recall, however, the following essential elements of the technique. In one class (typically an abstract superclass), there is a general method defined for a particular message that is inherited and used by subclasses. In at least one subclass, however, a method with the same name is defined, that hides access to the general method for instances of this class (or, in the case of refinement, subsumes access to the general method). We say the second method *overrides* the first.

Overriding is often transparent to the user of a class, and, as with overloading, frequently the two functions are thought of semantically as a single entity.

12.4.1 Replacement and Refinement

In Chapter 9 we briefly noted that overriding can occur in two different forms. A method can *replace* the method in the parent class, in which case the code in the parent is not executed at all. Alternatively, the code from the child can be used to form a *refinement*, which combines the code from the parent and the child classes.

Normally, overridden methods use replacement semantics. If a refinement is desired, it can be constructed by explicitly invoking the parent method as a function. This is accomplished by using the pseudo-variable `super` as the receiver in a message passing expression. An example from the Solitare program described in Chapter 9 showed this:

```
class DiscardPile extends CardPile {

    public void addCard (Card aCard) {
        if (! aCard.faceUp())
            aCard.flip();
        super.addCard(aCard);
        }

}
```

Constructors, on the other hand, *always* use refinement semantics. A constructor for a child class will always invoke the constructor for the parent class. This invocation will take place *before* the code for the constructor is executed. If the constructor for the parent class requires arguments, the pseudo-variable `super` is used as if it were a function:

```
class DeckPile extends CardPile {

    DeckPile (int x, int y) {
            // first initialize parent
        super(x, y);
            // then create the new deck
            // first put them into a local pile
        for (int i = 0; i < 4; i++)
            for (int j = 0; j <= 12; j++)
```

```
                addCard(new Card(i, j));

            // then shuffle the cards
        Random generator = new Random();
        for (int i = 0; i < 52; i++) {
            int j = Math.abs(generator.nextInt()) % 52;
                // swap the two card values
            Object temp = thePile.elementAt(i);
            thePile.setElementAt(thePile.elementAt(j), i);
            thePile.setElementAt(temp, j);
            }
        }

}
```

When used in this fashion, the call on the parent constructor must be the first statement executed. If no call on super is make explicitly and there exist two or more overloaded forms of the constructor, the constructor with no arguments (sometimes called the *default* constructor) will be the form used.

12.5 Abstract Methods

A method that is declared as *abstract* can be thought of as defining a method that is *deferred*; it is specified in the parent class but must be implemented in the child class. Interfaces can also be viewed as a method for defining deferred classes. Both can be considered to be a generalization of overriding. In both cases, the behavior described in a parent class is modified by the child class. In an abstract method, however, the behavior in the parent class is essentially null, a place holder, and *all* useful activity is defined as part of the code provided by the child class.

One advantage of abstract methods is conceptual, in that their use allows the programmer to think of an activity as associated with an abstraction at a higher level than may actually be the case. For example, in a collection of classes representing geometric shapes, we can define a method to draw the shape in each of the subclasses Circle, Square, and Triangle. We could have defined a similar method in the parent class Shape, but such a method cannot, in actuality, produce any useful behavior since the class Shape does not have sufficient information to draw the shape in question. Nevertheless, the mere presence of this method permits the user to associate the concept *draw* with the single class Shape, and not with the three separate concepts Square, Triangle, and Circle.

There is a second, more practical reason for using abstract methods. In statically typed object-oriented languages, such as Java, a programmer is permitted to send a message to an object only if the compiler can determine that there is in fact a corresponding method that matches the message selector. Suppose the programmer wishes to define a polymorphic variable of class Shape that will, at various times, contain instances of each of the different shapes. Such an assignment is possible, according to our rule of substitutability; nevertheless, the compiler will permit the message draw to be used with this variable only if it can ensure that the message will be understood by any value that may be associated with the variable. Assigning a method to the class Shape effectively provides this assurance, even when the method in class Shape is never actually executed.

12.6 Pure Polymorphism

Many authors reserve the term *polymorphism* (or *pure polymorphism*) for situations where one function can be used with a variety of arguments, and the term overloading for situations where there are multiple functions all defined with a single name.[2] Such facilities are not restricted to object-oriented languages. In Lisp or ML, for example, it is easy to write functions that manipulate lists of arbitrary elements; such functions are polymorphic, because the type of the argument is not known at the time the function is defined. The ability to form polymorphic functions is one of the most powerful techniques in object-oriented programming. It permits code to be written once, at a high level of abstraction, and to be tailored as necessary to fit a variety of situations. Usually, the programmer accomplishes this tailoring by sending further messages to the receiver for the method. These subsequent messages often are not associated with the class at the level of the polymorphic method, but rather are deferred methods defined in the lower classes.

An example will help us to illustrate this concept. As we noted in Chapter 8, the class Number is an abstract class, parent to the wrapper classes such as Integer, Double, Float. The definition of the class is similar to the following:

```
public abstract class Number {

    public abstract int intValue();

    public abstract long longValue();

    public abstract float floatValue();

    public abstract double doubleValue();
```

[2]The extreme cases may be easy to recognize, but discovering the line that separates overloading from polymorphism can be difficult. In both Java and ML a programmer can define a number of functions, each having the same name, but which take different arguments. Is it overloading in Java because the various functions sharing the same name are not defined in one location, whereas in ML-style polymorphism they must all be bundled together under a single heading?

```
    public byte byteValue()
        { return (byte) intValue(); }

    public short shortValue()
        { return (short) intValue(); }
}
```

The method intValue is abstract and deferred–each type of number must provide their own implementation of this method. The method byteValue, on the other hand, is not overridden. It is a purely polymorphic algorithm. Regardless of whether the receiver is an integer, a double precision floating point value, or some other type of number, this is the only definition that will be found. For all of these different types, when byteValue is invoked this will be the algorithm that is executed.

The important defining characteristic of pure polymorphism, as opposed to overloading and overriding, is that there is one function with the given name, used with a variety of different arguments. Almost always, as in this case, the body of such an algorithm will make use of other forms of polymorphism, such as the invocation of abstract functions shown here.

12.7 Efficiency and Polymorphism

An essential point to note is that programming always involves compromises. In particular, programming with polymorphism involves compromises between ease of development and use, readability, and efficiency. In large part, efficiency has been already considered and dismissed; however, it would be remiss not to admit that it is an issue, however slight.

A function, such as the byteValue method described in the last section, that does not know the type of its arguments can seldom be as efficient as a function that has more complete information. Nevertheless, the advantages of rapid development and consistent application behavior and the possibilities of code reuse usually more than make up for any small losses in efficiency.

12.8 Chapter Summary

Polymorphism is an umbrella term that is used to describe a variety of different mechanisms found in programming languages. In object-oriented languages the most important forms of polymorphism are tied to the polymorphic variable–a variable that can hold many different types of values. For example, overloading occurs when two or more functions share the same name. If these functions happen to be found in classes that have a parent class/child class relationship, then it is called overriding. If an overridden function is used with a polymorphic variable, then the particular function executed will be determined by the run-time value of the variable, not the compile-time declaration for the variable.

Other forms of polymorphism include overloading from independent classes, parameteric overloading (overloading that is disambiguated by the types of arguments used in a function call), and abstract methods.

Note that the use of polymorphism tends to optimize program development time and reliability, at the cost of run-time efficiency. For most programs, the benefits far exceed the costs.

Further Reading

In the interests of completeness, it should be mentioned that there is at least one important style of polymorphism, found in other computer languages, that is not found in Java. A *generic* (sometimes called a template) is a technique that allows a class description to be parameterized with a type. In C++, for example, one could declare a class as follows:

```
template <class T> class box {
public:
    box (T init) { value = initial; }
    T getValue() { return value; }
private
    T value;
};
```

The result is a "box of T", and not simply a box. To create such a value, one must also specify a type for the parameter value T:

```
box<int> aBox(5); // create a box with an integer
box<Shape> aBox(Circle); // create a box with a circle
```

One important place where this mechanism is useful is in the creation of collection classes (see Chapter 19). A language with generics, for example, would allow one to declare a vector *of Cards*, rather than (as in Java) simply a vector of objects. The compiler can then verify that the collection contains only the indicated type of values. More importantly, the compiler can avoid the cast necessary in Java when an object is removed from a container

A discussion of generics in relation to other forms of polymorphism can be found in [Budd 97].

Study Questions

1. What does the term polymorphic mean in common usage?

2. What is a polymorphic variable?

3. How is the characterization of polymorphic variables different in dynamically typed languages than in staticly typed languages?
4. What does it mean to say that a function name is overloaded?
5. What does it mean to say that a value has been coerced to a different type?
6. What is parameteric overloading?
7. What is overriding, and how is it different from overloading?
8. What is the difference between overriding using replacement, and overriding using refinement?
9. What is the default semantics for overriding for methods? For constructors?
10. What is an abstract method?
11. How is an abstract method denoted?
12. What characterizes pure polymorphism?
13. Why should a programmer not be overly concerned with the loss of efficiency due to the use of polymorphic programming techniques?

Exercises

1. Describe the various types of polymorphism found in the Pinball game application presented in Chapter 7.
2. Describe the various types of polymorphism found in the Solitare application presented in Chapter 9.