The first step in learning object-oriented programming is understanding the basic philosophy of organizing a computer program as the interaction of loosely coupled software components. This idea was the central lesson in the case studies presented in the first part of the book. The next step in learning object-oriented programming is organizing classes into a hierarchical structure based on the concept of inheritance. By inheritance, we mean the property that instances of a child class (or subclass) can access both data and behavior (methods) associated with a parent class (or superclass).

Although in Java the term inheritance is correctly applied only to the creation of new classes using subclassing (the extends keyword), there are numerous correspondences between subclassification and the designation that classes satisfy an interface (the implements keyword). The latter is sometimes termed “inheritance of specification,” contrasted with the “inheritance of code” provided by subclassification. In this chapter we will use the word in a general fashion, meaning both mechanisms.

Although the intuitive meaning of inheritance is clear, and we have used inheritance in many of our earlier case studies, and the mechanics of using inheritance are relatively simple, there are nevertheless subtle features involved in the use of inheritance in Java. In this and subsequent chapters we will explore some of these issues.

8.1 AN INTUITIVE DESCRIPTION OF INHERITANCE

Let us return to Flora the florist from the first chapter. There is a certain behavior we expect florists to perform, not because they are florists but simply because they are shopkeepers. For example, we expect Flora to request money for the transaction and in turn give us a receipt. These activities are not unique to
florists, but are common to bakers, grocers, stationers, car dealers, and other merchants. It is as though we have associated certain behavior with the general category Shopkeeper, and as Florists are a specialized form of shopkeepers, the behavior is automatically identified with the subclass.

In programming languages, inheritance means that the behavior and data associated with child classes are always an extension (that is, a larger set) of the properties associated with parent classes. A child class will be given all the properties of the parent class, and may in addition define new properties. On the other hand, since a child class is a more specialized (or restricted) form of the parent class, it is also, in a certain sense, a contraction of the parent type. For example, the Java library Frame represents any type of window, but a PinballGame frame is restricted to a single type of game. This tension between inheritance as expansion and inheritance as contraction is a source for much of the power inherent in the technique, but at the same time it causes much confusion as to its proper employment. We will see this when we examine a few of the uses of inheritance in a subsequent section.

Inheritance is always transitive, so that a class can inherit features from superclasses many levels away. That is, if class Dog is a subclass of class Mammal, and class Mammal is a subclass of class Animal, then Dog will inherit attributes both from Mammal and from Animal.

A complicating factor in our intuitive description of inheritance is the fact that subclasses can override behavior inherited from parent classes. For example, the class Platypus overrides the reproduction behavior inherited from class Mammal, since platypuses lay eggs. The mechanics of overriding is treated briefly in this chapter and in more detail in Chapter 11.

### 8.2 The Base Class Object

In Java all classes use inheritance. Unless specified otherwise, all classes are derived from a single root class, named Object. If no parent class is explicitly provided, the class Object is implicitly assumed. Thus, the class declaration for FirstProgram (Chapter 4, Figure 4.1) is the same as the following:

```java
class FirstProgram extends Object {
    // ...
};
```

The class Object provides minimal functionality guaranteed to be common to all objects. These include the following methods:

- `equals(Object obj)` Determine whether the argument object is the same as the receiver. This method is often overridden to change the equality test for different classes.

- `getClass()` Returns the class of the receiver, an object of type Class (see Section 13.5).
8.3 Subclass, Subtype, and Substitutability

hashCode() Returns a hash value for this object (see Section 19.7). This method should also be overridden when the equals method is changed.

toString() Converts object into a string value. This method is also often overridden.

8.3 Subclass, Subtype, and Substitutability

The concept of substitutability is fundamental to many of the most powerful software development techniques in object-oriented programming. The idea of substitutability is that the type given in a declaration of a variable does not have to match the type associated with a value the variable is holding. Note that this is never true in conventional programming languages but is a common occurrence in object-oriented programs.

We have seen several examples of substitutability in our earlier case studies. In the Pin Ball Game program described in Chapter 7, the variable target was declared as a PinBallTarget, but in fact held a variety of different types of values that were created using implementations of PinBallTarget. (These target values were held in the vector named targets.)

PinBallTarget target = (PinBallTarget) targets.elementAt(j);

Substitutability can also occur through the use of interfaces. An example is the instance of the class FireButtonListener created in the Cannon Game (Chapter 6). The class from which this value was defined was declared as implementing the interface ActionListener. Because it implements the ActionListener interface, we can use this value as a parameter to a method (in this case, addActionListener) that expects an ActionListener value.

class CannonWorld extends Frame {
    
    private class FireButtonListener implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            ...
        }
    }

    public CannonWorld () {
        ...
        fire.addActionListener(new FireButtonListener());
    }
}
Because Object is a parent class to all objects, a variable declared using this type can hold any nonprimitive value. The collection class Vector makes use of this property, holding its values in an array of Object values. Because the array is declared as Object, any object value can be stored in a Vector.

When new classes are constructed using inheritance from existing classes, the argument used to justify the validity of substitutability is as follows:

- Instances of the subclass must possess all data fields associated with the parent class.
- Instances of the subclass must implement, through inheritance at least (if not explicitly overridden) all functionality defined for the parent class. (They can also define new functionality, but that is unimportant for the argument.)
- Thus, an instance of a child class can mimic the behavior of the parent class and should be indistinguishable from an instance of the parent class if substituted in a similar situation.

We will see later in this chapter, when we examine the various ways in which inheritance can be used, that this is not always a valid argument. Thus, not all subclasses formed using inheritance are candidates for substitution.

The term subtype is used to describe the relationship between types that explicitly recognizes the principle of substitution. That is, a type B is considered to be a subtype of A if two conditions hold. The first is that an instance of B can legally be assigned to a variable declared as type A. And the second is that this value can then be used by the variable with no observable change in behavior.

The term subclass refers merely to the mechanics of constructing a new class using inheritance, and is easy to recognize from the source description of a program by the presence of the keyword extends. The subtype relationship is more abstract, and is only loosely documented directly by the program source. In the majority of situations a subclass is also a subtype. However, later in this chapter we will discover ways in which subclasses can be formed that are not subtypes. In addition, subtypes can be formed using interfaces, linking types that have no inheritance relationship whatsoever. So it is important to understand both the similarities and the differences between these two concepts.

### 8.4 Forms of Inheritance

Inheritance is employed in a surprising variety of ways. Presented in this section are a few of its more common uses. Note that the following list represents general abstract categories and is not intended to be exhaustive. Furthermore, it sometimes happens that two or more descriptions are applicable to a single situation, because some methods in a single class use inheritance in one w
8.4 Forms of Inheritance

while others use it in another. In the following list, pay attention to which uses of inheritance support the subtyping relationship and which do not.

8.4.1 Inheritance for Specialization

Probably the most common use of inheritance and subclassification is for specialization. In this form, the new class is a specialized variety of the parent class but it satisfies the specifications of the parent in all relevant respects. Thus, this form always creates a subtype, and the principle of substitutability is explicitly upheld. Along with the following category (subclassification for specification) this is the most ideal form of inheritance, and something that a good design should strive for.

The creation of application window classes using inheritance from the Java library class Frame is an example of subclassification for specialization. The following is from the PinBallGame program in Chapter 7.

```java
public class PinBallGame extends Frame {
    ...
}
```

To run such an application, an instance of PinBallGame is first created. Various methods inherited from class Frame, such as setSize, setTitle, and show, are then invoked. These methods do not realize they are manipulating an instance of PinBallGame, but instead act as if they were operating on an instance of Frame. The actions they perform would be the same for any instance of class Frame.

Where application-specific behavior is necessary, for example, in repainting the window, a method is invoked that is overridden by the application class. For example, the method in the parent class will invoke the method paint. Although the parent class Frame possesses a method of this name, the parent method is not the one executed. Instead, the method defined in the child class is executed.

We say that subclassification for specialization occurs in this example because the child class (in this example, PinBallGame) satisfies all the properties that we expect of the parent class (Frame). In addition, the new class overrides one or more methods, specializing them with application-specific behavior.

8.4.2 Inheritance for Specification

Another frequent use for inheritance is to guarantee that classes maintain a certain common interface—that is, they implement methods having the same headings. The parent class can be a combination of implemented operations and operations that are deferred to the child classes. Often, there is no interface change of any sort between the parent class and the child class—the child merely implements the methods described, but not implemented, in the parent.
UNDERSTANDING INHERITANCE

This is actually a special case of subclassification for specialization, except that the subclasses are not refinements of an existing type but rather realizations of an incomplete abstract specification. That is, the parent class defines the operation but has no implementation. It is only the child class that provides an implementation. In such cases the parent class is sometimes known as an abstract specification class.

There are two different mechanisms provided by the Java language to support the idea of inheritance of specification. The most obvious technique is the use of interfaces. We have seen examples of this in the way that events are handled by the Java library. For instance, the characteristics needed for an ActionListener (the object type that responds to button presses) can be described by a single method, and the implementation of that method cannot be predicted, since it differs from one application to another. Thus, an interface is used to describe only the necessary requirements, and no actual behavior is inherited by a subclass that implements the behavior.

```
interface ActionListener {
    public void actionPerformed (ActionEvent e);
}
```

When a button is created, an associated listener class is defined. The listener class provides the specific behavior for the method in the context of the current application.

```
class CannonWorld extends Frame {
    
    // a fire button listener implements the action listener interface
    private class FireButtonListener implements ActionListener {
        public void actionPerformed (ActionEvent e) {
            ... // action to perform in response to button press
        }
    }

    
}
```

Subclassification for specification can also take place with inheritance of classes formed using extension. One way to guarantee that a subclass must be constructed is to use the keyword abstract. A class declared as abstract must be subclassed; it is not possible to create an instance of such a class using the operator new. In addition, individual methods can also be declared as abstract, and they, too, must be overridden before instances can be constructed.

An example abstract class in the Java library is Number, a parent class for the numeric wrapper classes Integer, Long, Double, and so on. The class description is as follows:
public abstract class Number {
    public abstract int intValue();
    public abstract long longValue();
    public abstract float floatValue();
    public abstract double doubleValue();
    public byte byteValue() {
        return (byte) intValue();
    }
    public short shortValue() {
        return (short) intValue();
    }
}

Subclasses of Number must override the methods intValue, longValue, floatValue, and doubleValue. Notice that not all methods in an abstract class must themselves be declared abstract. Subclasses of Number need not override byteValue or shortValue, as these methods are provided with an implementation that can be inherited without change.

In general, subclassification for specification can be recognized when the parent class does not implement actual behavior but merely provides the headings for methods that must be implemented in child classes.

### 8.4.3 Inheritance for Construction

A class can often inherit almost all of its desired functionality from a parent class, perhaps changing only the names of the methods used to interface to the class, or modifying the arguments. This may be true even if the new class and the parent class fail to share any relationship as abstract concepts.

An example of subclassification for construction occurred in the pinball game application described in Chapter 7. In that program, the class Hole was declared as a subclass of Ball. There is no logical relationship between the concepts of a Ball and a Hole, but from a practical point of view much of the behavior needed for the Hole abstraction matches the behavior of the class Ball. Thus, using inheritance in this situation reduces the amount of work necessary to develop the class Hole.

```java
class Hole extends Ball implements PinBallTarget {
    public Hole (int x, int y) {
        super (x, y, 12);
        setColor (Color.black);
    }
```
public boolean intersects (Ball aBall)
    { return location.intersects(aBall.location); }

public void hitBy (Ball aBall) {
    // move ball totally off frame
    aBall.moveTo (0, PinBallGame.FrameHeight + 30);  // stop motion of ball
    aBall.setMotion(0, 0);
}

Another example of inheritance for construction occurs in the Java library. There, the class Stack is constructed using inheritance from the class Vector:

class Stack extends Vector {

    public Object push(Object item)
        { addElement(item); return item; }

    public boolean empty ()
        { return isEmpty(); }

    public synchronized Object pop()
        { Object obj = peek();
          removeElementAt(size() - 1);
          return obj;
        }

    public synchronized Object peek()
        { return elementAt(size() - 1); }

}

As abstractions, the concept of the stack and the concept of a vector have little in common; however, from a pragmatic point of view using the Vector class as a parent greatly simplifies the implementation of the stack.

Inheritance for construction is sometimes frowned upon, since it often directly breaks the principle of substitutability (forming subclasses that are not subtypes). On the other hand, because it is often a fast and easy route to developing new data abstractions, it is nevertheless widely used. See Chapter 10 for a more detailed discussion of the construction of the Stack abstraction.
8.4.4 Inheritance for Extension

Subclassification for extension occurs when a child class only adds new behavior to the parent class and does not modify or alter any of the inherited attributes. An example of inheritance for extension in the Java library is the class Properties, which inherits from class HashTable. A hash table is a dictionary structure (see Section 19.7). A dictionary stores a collection of key/value pairs and allows the user to retrieve the value associated with a given key. Properties represent information concerning the current execution environment. Examples of properties are the name of the user running the Java program, the version of the Java interpreter being used, the name of the operating system under which the Java program is running, and so on. The class Properties uses the parent class, HashTable, to store and retrieve the actual property name/value pairs. In addition, the class defines a few methods specific to the task of managing properties, such as reading or writing properties to or from a file.

```java
class Properties extends Hashtable {
    
    public synchronized void load(InputStream in) throws IOException { ... }

    public synchronized void save(OutputStream out, String header) { ... }

    public String getProperty(String key) { ... }

    public Enumeration propertyNames() { ... }

    public void list(PrintStream out) { ... }
}
```

As the functionality of the parent remains available and untouched, subclassification for extension does not contravene the principle of substitutability, and so such subclasses are always subtypes.

8.4.5 Inheritance for Limitation

Subclassification for limitation occurs when the behavior of the subclass is smaller or more restrictive than the behavior of the parent class. Like subclassification for extension, subclassification for limitation occurs most frequently when a programmer is building on a base of existing classes that should not, or cannot, be modified.
Although there are no examples of subclassification for limitation in the Java library, we could imagine the following. Suppose you wanted to create the class Set, in a fashion similar to the way the class Stack is subclassed from Vector. Say you also wanted to ensure that only Set operations were used on the set, and not vector operations. One way to accomplish this would be to override the undesired methods, so that if they were executed they would produce obviously incorrect results, or print a message indicating they should not be used.\(^1\)

```java
class Set extends Vector {
    // methods addElement, removeElement, contains
    // isEmpty and size
    // are all inherited from vector

    public int indexOf(Object obj)
    { System.out.println("Do not use Set.indexOf"); return 0; }

    public Object elementAt(int index)
    { return null; }
}
```

In theory an alternative would be to have the undesired methods throw an exception. However, the Java compiler does not permit subclasses to override a method and introduce new exceptions that are not already declared in the parent class.

Subclassification for limitation is characterized by the presence of techniques that take a previously permitted method and make it illegal. Because subclassification for limitation is an explicit contravention of the principle of substitutability, and because it builds subclasses that are not subtypes, it should be avoided whenever possible.

### 8.4.6 Inheritance for Combination

When discussing abstract concepts, it is common to form a new abstraction by combining features of two or more abstractions. A teaching assistant, for example, may have characteristics of both a teacher and a student, and can therefore logically behave as both. The ability of a class to inherit from two or more parent classes is known as multiple inheritance.

Although the Java language does not permit a subclass to be formed by inheritance from more than one parent class, several approximations to the concept are possible. For example, it is common for a new class to both extend an existing class and implement an interface. We saw this in the example of

---

\(^1\) In actuality, the methods indexOf and elementAt are declared as final in class Vector, so this example will not compile. But it does illustrate the concept.
the class Hole that both extended class Ball and implemented the interface for PinBallTarget.

```java
class Hole extends Ball implements PinBallTarget {
    
}
```

It is also possible for classes to implement more than one interface, and thus be viewed as a combination of the two categories. Many examples occur in the input/output sections of the Java library. A RandomAccessFile, for example, implements both the DataInput and DataOutput protocols.

### 8.4.7 Summary of the Forms of Inheritance

We can summarize the various forms of inheritance by the following list:

- **Specialization** The child class is a special case of the parent class; in other words, the child class is a subtype of the parent class.

- **Specification** The parent class defines behavior that is implemented in the child class but not in the parent class.

- **Construction** The child class makes use of the behavior provided by the parent class but is not a subtype of the parent class.

- **Extension** The child class adds new functionality to the parent class, but does not change any inherited behavior.

- **Limitation** The child class restricts the use of some of the behavior inherited from the parent class.

- **Combination** The child class inherits features from more than one parent class. Although multiple inheritance is not supported directly by Java, it can be simulated in part by classes that use both inheritance and implementation of an interface, or implement two or more interfaces.

The Java language implicitly assumes that subclasses are also subtypes. This means that an instance of a subclass can be assigned to a variable declared as the parent class type. Methods in the child class that have the same name as those in the parent class override the inherited behavior. We have seen that this assumption that subclasses are subtypes is not always valid, and creating subclasses that are not subtypes is a possible source of program error.