Chapter 8

Understanding Inheritance

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 subclassing 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 subclassing. In this chapter we will use the word in a general fashion, meaning both mechanisms.

While 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. We will briefly mention the mechanics of overriding in this chapter, then return to a more detailed discussion of the semantics of overriding 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:

```
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 name of the class of the receiver as a string.

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 may not 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 subclasses 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-ball 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 function (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 non-primitive 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 areas 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. In this section we will describe 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 sometime happens that two or more descriptions are applicable to a single situation, because some methods in a single class use inheritance in one way while others use it in another. In the following list, pay careful 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 subclassing is for specialization. In this form, the new class is a specialized variety of the parent class but 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 (subclassing 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 subclassing for specialization. The following is from the PinBall Game program in Chapter 7.

```
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 repaint. Although the parent class Frame possesses a method of this name, the parent method is not the one executed. Instead, the function defined in the child class is executed.

We say that subclassing for specialization is occurring 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 the same methods. 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 behavior described, but not implemented, in the parent.

This is actually a special case of subclassing 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
        }
    }
}
```

Subclassing 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, subclassing for specification can be recognized when the parent class does not implement actual behavior but merely defines the behavior 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 subclassing for construction occurred in the Pin ball 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.

```
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. In Chapter 10 we will discuss the construction of the Stack abstraction in more detail.

8.4.4 Inheritance for Extension

Subclassing 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.

```
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, subclassing for extension does not contravene the principle of substitutability and so such subclasses are always subtypes.

8.4.5 Inheritance for Limitation

Subclassing for limitation occurs when the behavior of the subclass is smaller or more restrictive than the behavior of the parent class. Like subclassing for extension, subclassing for limitation occurs most frequently when a programmer is building on a base of existing classes that should not, or cannot, be modified.

There are no examples of subclassing for limitation in the Java library, however we could imagine the following. Suppose one wanted to create the class Set, in a fashion similar to the way the class Stack is subclassed from Vector. However, 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 generate an exception.[1]

```
class Set extends Vector {
    // methods addElement, removeElement, contains
```

[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.

```
    // isEmpty and size
    // are all inherited from vector

    public int indexOf (Object obj)
        { throw new IllegalOperation("indexOf"); }

    public int elementAt (int index)
        { throw new IllegalOperation("indexOf"); }
}
```

Where IllegalOperation is a subclass of Exception:

```
class IllegalOperation extends Exception {
    IllegalOperation (String str) { super(str); }
    }
```

Subclassing for limitation is characterized by the presence of methods that take a previously permitted operation and makes it illegal. Because subclassing 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 discussion abstract concepts, it is common for a new abstraction to be formed as a combination of features from 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 the class Hole that both extended class Ball and implemented the interface for PinBallTarget.

```
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 table:

- 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.

8.5 Modifiers and Inheritance

The language Java provides several modifiers that can be used to alter aspects of the inheritance process. For example, in the case studies in earlier chapters, we made extensive use of the visibility (or access control) modifiers public, protected and private.

- A public feature (data field or method) can be accessed outside the class definition. A public class can be accessed outside the package in which it is declared.
- A protected feature can be accessed only within the class definition in which it appears, or within the definition of subclasses.
- A private feature can be accessed only within the class definition in which it appears.

We have seen from our first case studies how both methods and data fields can be declared as static. A static field is shared by all instances of a class. A static method can be invoked even when no instance of the class has been created. Static data fields and methods are inherited in the same manner as non-static items, except that static methods cannot be overridden.

Both methods and classes can be declared to be abstract. An abstract class cannot be instanciated. That is, it is not legal to create an instance of an abstraction class using the operator new. Such a class can only be used as a parent class, to create a new type of object. Similarly, an abstract method must be overridden by a subclass.

An alternative modifier, final, is the opposite of abstract. When applied to a class, the keyword indicates that the class *cannot* be subclassed. Similarly, when applied to a method, the keyword indicates that the method cannot be overridden. Thus, the user is guaranteed that the behavior of the class will be as defined and not modified by a later subclass.

```
final class newClass extends oldClass {
    ...
    }
```

We have seen that program constants are generally defined by variables that are both static and final:

```
class CannonGame extends Frame {
    ...
    public static final int FrameWidth = 600;
    public static final int FrameHeight = 400;
    ...
}
```

Optimizing compilers can sometimes make use of the fact that a data field, class or method is declared as final, and generate better code than would otherwise be possible.

8.6 Programming as a Multi Person Activity

When programs are constructed out of reusable, off-the-shelf components, programming moves from an individual activity (one programmer and the computer) to a community effort. A programmer may operate both as the *developer* of new abstractions, and as the *user* of a software system created by an earlier programmer. The reader should not confuse the term *user* when applied to a programmer with the same term denoting the application end-user. Similarly, we will often speak of the organization of several objects by describing a *client* object, that is requesting the services of a *provider*. Again, the client in this case is likely a programmer (or the code being developed by a programmer) making use of the services developed by an earlier programmer. This should not be confused with the idea of *client/sever* computing, as described in Chapter 2.

8.7 The Benefits of Inheritance

In this section we will describe some of the many important benefits of the proper use of inheritance.

8.7.1 Software Reusability

When behavior is inherited from another class, the code that provides that behavior does not have to be rewritten. This may seem obvious, but the implications are important. Many programmers spend much of their time rewriting code they have written many times before–for example, to search for a pattern in a string or to insert a new element into a table. With object-oriented techniques, these functions can be written once and reused.

8.7.2 Increased Reliability

Code that is executed frequently will tend to have fewer bugs then code that executed infrequently. When the same components are used in two or more applications, the code will be exercised more than code that is developed for a single application. Thus, bugs in such code tend to be more quickly discovered, and latter applications gain the benefit of using components are more error free. Similarly, the costs of maintenance of shared components can be split among many projects.

8.7.3 Code Sharing

Code sharing can occur on several levels with object-oriented techniques. On one level, many users or projects can use the same classes. (Brad Cox [Cox 1986] calls these software-ICs, in analogy to the integrated circuits used in hardware design). Another form of sharing occurs when two or more classes developed by a single programmer as part of a project inherit from a single parent class. For example, a Set and an Array may both be considered a form of Collection. When this happens, two or more types of objects will share the code that they inherit. This code needs to be written only once and will contribute only once to the size of the resulting program.

8.7.4 Consistency of Interface

When two or more classes inherit from the same superclass, we are assured that the behavior they inherit will be the same in all cases. Thus, it is easier to guarantee that interfaces to similar objects are in fact similar, and that the user is not presented with a confusing collection of objects that are almost the same but behave, and are interacted with, very differently.

8.7.5 Software Components

Inheritance provides programmers with the ability to construct reusable software components. The goal is to permit the development of new and novel applications that nevertheless require little or no actual coding. The Java library provides a rich collection of software components for use in the development of applications.

8.7.6 Rapid Prototyping

When a software system is constructed largely out of reusable components, development time can be concentrated on understanding the new and unusual portion of the system. Thus, software systems can be generated more quickly and easily, leading to a style of programming known as *rapid prototyping* or *exploratory programming*. A prototype system is developed, users experiment with it, a second system is produced that is based on experience with the first, further experimentation takes place, and so on for several iterations. Such programming is particularly useful in situations where the goals and requirements of the system are only vaguely understood when the project begins.

8.7.7 Polymorphism and Frameworks

Software produced conventionally is generally written from the bottom up, although it may be *designed* from the top down. That is, the lower-level routines are written, and on top of these slightly higher abstractions are produced, and on top of these even more abstract elements are generated. This process is like building a wall, where every brick must be laid on top of an already laid brick.

Normally, code portability decreases as one moves up the levels of abstraction. That is, the lowest-level routines may be used in several different projects, and perhaps even the next level of abstraction may be reused, but the higher-level routines are intimately tied to a particular application. The lower-level pieces can be carried to a new system and generally make sense standing on their own; the higher-level components generally make sense (because of declarations or data dependencies) only when they are built on top of specific lower-level units.

Polymorphism in programming languages permits the programmer to generate high-level reusable components that can be tailored to fit different applications by changes in their low-level parts. The Java AWT is an example of a large software framework that relies on inheritance and substitutability for its operation.

8.7.8 Information Hiding

A programmer who reuses a software component needs only to understand the nature of the component and its interface. It is not necessary for the programmer to have detailed information concerning matters such as the techniques used to implement the component. Thus, the interconnectedness between software systems is reduced. We earlier identified the interconnected nature of conventional software as being one of the principle causes of software complexity.

8.8 The Costs of Inheritance

Although the benefits of inheritance in object-oriented programming are great, almost nothing is without cost of one sort or another. For this reason, we must consider the cost of object-oriented programming techniques, and in particular the cost of inheritance.

8.8.1 Execution Speed

It is seldom possible for general-purpose software tools to be as fast as carefully handcrafted systems. Thus, inherited methods, which must deal with arbitrary subclasses, are often slower than specialized code.

Yet, concern about efficiency is often misplaced.[2] First, the difference is often small. Second, the reduction in execution speed may be balanced by an increase in the speed of software development. Finally, most programmers actually have little idea of how execution time is being used in their programs. It is far better to develop a working system, monitor it to discover where execution time is being used, and improve those sections, than to spend an inordinate amount of time worrying about efficiency early in a project.

8.8.2 Program Size

The use of any software library frequently imposes a size penalty not imposed by systems constructed for a specific project. Although this expense may be substantial, as memory costs decrease the size of programs becomes less important. Containing development costs and producing high-quality and error-free code rapidly are now more important than limiting the size of programs.

8.8.3 Message-Passing Overhead

Much has been made of the fact that message passing is by nature a more costly operation than simple procedure invocation. As with overall execution speed, however, overconcern

[2]The following quote from an article by Bill Wulf offers some apt remarks on the importance of efficiency: “More computing sins are committed in the name of efficiency (without necessarily achieving it) than for any other single reason–including blind stupidity” [Wulf 1972].

about the cost of message passing is frequently penny-wise and pound-foolish. For one thing, the increased cost is often marginal–perhaps two or three additional assembly-language instructions and a total time penalty of 10 percent. This increased cost, like others, must be weighed against the many benefits of the object-oriented technique.

8.8.4 Program Complexity

Although object-oriented programming is often touted as a solution to software complexity, in fact, overuse of inheritance can often simply replace one form of complexity with another. Understanding the control flow of a program that uses inheritance may require several multiple scans up and down the inheritance graph. This is what is known as the *yo-yo* problem, which we will discuss in more detail in a later chapter.

8.9 Chapter Summary

Inheritance is a mechanism for relating a new software abstraction being developed to an older, existing abstraction. By stating that the new component inherits (or *extends*) the older abstraction, the programmer means that all the public and protected properties of the original class are also now part of the new abstraction. In addition, the new class can add new data fields and behavior, and can override methods that are inherited from the original class. Interfaces are a closely related mechanism, which tie the concrete realization of behavior to an abstract description.

All classes in Java use inheritance. If not explicitly stated, classes are assumed to inherit from the fundamental root class Object.

Inheritance is tied to the principle of substitutability. A variable that is declared as one class can be assigned a value that created from a child class. A similar mechanism also works with interfaces. A class that can be used in lieu of another class is said to be a *subtype*. Java implicitly assumes that all subclasses are subtypes. However, this need not be true (a subclass can override a method in an incompatible fashion, for example). Subtypes can also be constructed from interfaces, avoiding subclasses altogether.

There are many different types of inheritance, used for different purposes. Variations include specialization, specification, construction, extension, limitation, and combination.

A variety of modifiers alter the meaning of inheritance. A private feature is not inherited by subclasses. A static feature (data field or method) is shared by all instances. An abstract method must be overridden. A final feature (data field or method) cannot be overridden.

Study Questions

1. Give an intuitive description of inheritance.

2. What does it mean for a method to override an inherited method?

3. What is the name of the root class for all objects in Java?

4. What behavior is provided by the root class in Java?

5. What does it mean to say that child classes are substitutable for parent classes in Java?

6. What is the difference between a subclass and a subtype?

7. What are the characteristics of inheritance for specialization?

8. What are the characteristics of inheritance for specification? How does this differ from inheritance for specialization?

9. What are the characteristics of inheritance for construction? Why is this not generally considered to be a good use of inheritance?

10. What are the characteristics of inheritance for extension?

11. What are the characteristics of inheritance for limitation? Why is this not generally considered to be a good use of inheritance?

12. Why would it not make sense for a method in Java to be declared both abstract and final ?

13. What are some of the benefits of developing classes using inheritance, rather than developing each new class from scratch?

14. What are some of the costs of using inheritance for software development?

Exercises

1. Suppose you were required to program a project in a non-object oriented language, such as Pascal or C. How would you simulate the notion of classes and methods? How would you simulate inheritance? Could you support multiple inheritance? Explain your answer.

2. We noted that the execution overhead associated with message passing is typically greater than the overhead associated with a conventional procedure call. How might you measure these overheads? For a language that supports both classes and procedures (such as C++ or Object Pascal), devise an experiment to determine the actual performance penalty of message passing.

3. Consider the three geometric concepts of a line (infinite in both directions), a ray (fixed at a point, infinite in one direction), and a segment (a portion of a line with fixed end points). How might you structure classes representing these three concepts in an inheritance hierarchy? Would your answer differ if you concentrated more on the data representation or more on the behavior? Characterize the type of inheritance you would use. Explain the reasoning behind your design.

4. Why is the example used in the following explanation not a valid illustration of inheritance?

> Perhaps the most powerful concept in object-oriented programming systems is inheritance. Objects can be created by inheriting the properties of other objects, thus removing the need to write any code whatsoever! Suppose, for example, a program is to process complex numbers consisting of real and imaginary parts. In a complex number, the real and imaginary parts behave like real numbers, so all of the operations (+, -, /, *, sqrt, sin, cos, etc.) can be inherited from the class of objects call REAL, instead of having to be written in code. This has a major impact on programmer productivity.