Chapter 9

Inheritance and Interfaces

Introduction

In this chapter we will explore two of the most important techniques the Java language provides to help you write better structured solutions. Inheritance allows you to share code between classes to reduce redundancy and lets you treat different classes of objects in the same way. Interfaces allow you to treat several different classes of objects the same way without sharing code.

It is difficult to show the usefulness of inheritance and interfaces in simple examples, but in larger and more complex projects these techniques are invaluable. The Java class libraries make extensive use of these two features. Inheritance and interfaces make it possible to create well-structured code on a scale that is as large as the entire Java class libraries.

9.1 Inheritance Basics

We'll begin our discussion of inheritance by exploring how the concept originated and considering a nonprogramming example that will lead us toward programming with inheritance in Java.

Large programs demand that we write versatile and clear code on a large scale. In this textbook, we've examined several ways to express programs more concisely and elegantly on a small scale. Features like static methods, parameterization, loops, and classes help us organize our programs and extract common features that can be used in many places. This general practice is called *code reuse.*

> **Code Reuse**
>
> The practice of writing program code once and using it in many contexts.

Did You Know?

The Software Crisis

Software has been getting more and more complicated since the advent of programming. By the early 1970s, teams writing larger and more complex programs began to encounter some common problems. Despite much effort, software projects were running over budget and were not being completed on time; also, the software often had bugs, didn't do what it was supposed to do, or was otherwise of low quality. In his 1975 book *The Mythical Man-Month: Essays on Software Engineering*, software engineer Fred Brooks argued that adding manpower to a late software project often made it finish even later. Collectively, these problems came to be called the "software crisis."

A particularly sticky issue involved program maintenance. Companies found that they spent much of their time not writing new code but modifying and maintaining existing code (also called *legacy code*). This proved to be a difficult task, because it was easy to write disorganized and redundant code. Maintenance of such code was likely to take a long time and to introduce new bugs into the system.

The negative effects of the software crisis and maintenance programming were particularly noticeable when graphical user interfaces became prominent in the 1980s. User interfaces in graphical systems like Microsoft Windows and Apple's Mac OS X were much more sophisticated than the text interfaces that preceded them. The original graphical programs were prone to redundancy because they had to describe in detail how to implement buttons, text boxes, and other onscreen components. Also, the graphical components themselves contained a lot of common states and behavior, such as particular sizes, shapes, colors, positions, or scrollbars.

Object-oriented programming provides us with a feature called inheritance that increases our ability to reuse code by allowing one class to be an extension of another. Inheritance also allows us to write programs with hierarchies of related object types.

Nonprogramming Hierarchies

In order to use inheritance, you'll want to identify similarities between different objects and classes in your programs. Let's start by looking at a nonprogramming example: a hierarchy of employees at a company.

Imagine a large law firm that hires several classes of employees: lawyers, general secretaries, legal secretaries, and marketers. The company has a number of employee rules about vacation and sick days, medical benefits, harassment regulations, and so on. Each subdivision of the company also has a few of its own rules; for example, lawyers may use a different form to ask for vacation leave than do secretaries.

Suppose that all the employees attend a common orientation where they learn the general rules. Each employee receives a 20-page manual of these rules to read. A mixed group of employees could attend the orientation together: Lawyers, secretaries, and marketers all might sit in the same orientation group.

Afterward, the employees go to their subdivisions and receive secondary, smaller orientations covering any rules specific to those divisions. Each employee receives a smaller manual, two or three pages in length, covering that subdivision's specific rules. Some rules are added to those in the general 20-page manual, and a few are replaced. For example, one class of employees may get three weeks of vacation instead of two, and one class may use a pink form to apply for time off rather than the yellow form listed in the 20-page manual. Each class has its own submanual with unique contents as shown in Figure 9.1.

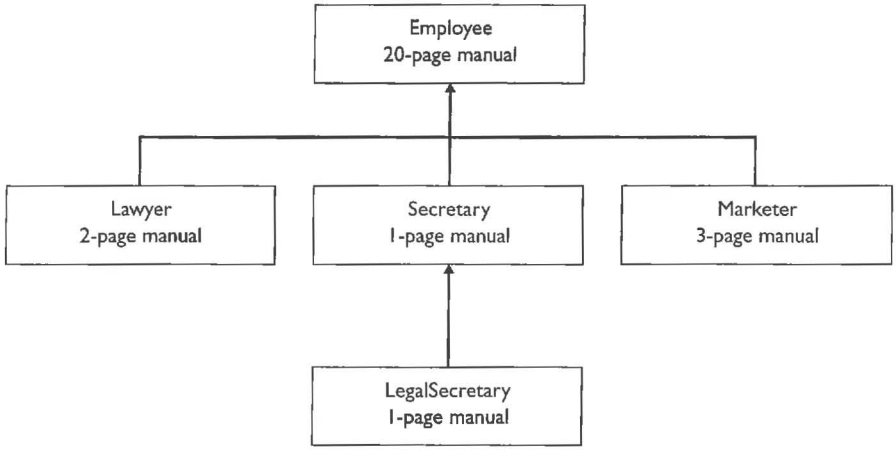


Figure 9.1 A hierarchy of employee manuals

An alternative solution would be to give every employee a large manual containing both the applicable general rules and the rules of their subdivisions. For example, there might be a 22-page manual for the lawyers, a 21-page manual for secretaries, and a 23-page manual for marketers. The consolidation might even save a few pages. So why does the company bother to generate two manuals for every employee?

The main reason has to do with redundancy and maintenance. The 22-page lawyer manual contains a lot of the same text as the 21-page secretary manual. If a common rule is changed under the one-manual scenario, all the manuals need to be updated individually, which is a tedious process. Making the same change to many manuals is also likely to introduce errors, because it is easy to make the change in one copy but forget to do it in another.

In addition, there's a certain practical appeal to using the shorter, more specific manuals. Someone who wants to know all the rules that are specific to lawyers can simply read the 2-page lawyer manual, rather than combing through a 22-page lawyer manual trying to spot differences.

There are two key ideas here:

1. It's useful to be able to specify a broad set of rules that will apply to many related groups (the 20-page manual).
2. It's also useful to be able to specify a smaller set of rules specific to a particular group, and to be able to replace some rules from the broad set (e.g., "use the pink form instead of the yellow form").

An important thing to notice about the categories is that they are hierarchical. For example, every legal secretary is also a secretary, and every marketer is also an employee. In a pinch, you could ask a legal secretary to work as a standard secretary for a short period, because a legal secretary is a secretary. We call such a connection an *is-a relationship.*

Is-a Relationship

A hierarchical connection between two categories in which one type is a specialized version of the other.

An is-a relationship is similar to the idea of a role. A legal secretary can also fill the roles of a secretary and an employee. A lawyer can also fill the role of an employee. A member of a subcategory can add to or change behavior from the larger category. For example, a legal secretary adds the ability to file legal briefs to the secretary role and may change the way in which dictation is taken.

Each group of employees in our example is analogous to a class in programming. The different employee groups represent a set of related classes connected by is-a relationships. We call such a set of classes an *inheritance hierarchy.*

Inheritance Hierarchy

A set of hierarchical relationships between classes of objects.

As you'll see, inheritance hierarchies are commonly used in Java to group related classes of objects and reuse code between them.

Extending a Class

The previous section presented a nonprogramming example of hierarchies. But as an exercise, we could write small Java classes to represent those categories of employees. The code will be a bit silly but will illustrate some important concepts.

Let's imagine that we have the following rules for our employees:

- Employees work 40 hours per week.
- All employees earn a salary of $40,000 per year, with the exception of marketers, who make $50,000 per year, and legal secretaries, who make $45,000 per year.
- Employees have two weeks of paid vacation leave per year, with the exception of lawyers, who have three weeks of vacation leave.
- Employees use a yellow form to apply for vacation leave, with the exception of lawyers, who use a special pink form.
- Each type of employee has unique behavior: Lawyers know how to handle lawsuits, marketers know how to advertise, secretaries know how to take dictation, and legal secretaries know how to file legal briefs.

Let's write a class to represent the common behavior of all employees. (Think of this as the 20-page employee manual.) We'll write methods called `getHours`, `getSalary`, `getVacationDays`, and `getVacationForm` to represent these behaviors. To keep things simple, each method will just return some value representing the default employee behavior, such as the $40,000 salary and the yellow form for vacation leave. We won't declare any fields for now. Here is the code for the basic `Employee` class:

```
1  // A class to represent employees in general.
2  public class Employee {
3      public int getHours() {
4          return 40;
5      }
6
7      public double getSalary() {
8          return 40000.0;
9      }
10
11     public int getVacationDays() {
12         return 10;
13     }
14
```

```
15      public String getVacationForm() {
16          return "yellow";
17      }
18  }
```

Now let's think about implementing the `Secretary` subcategory. As we mentioned in the previous section, every `Secretary` is also an `Employee` and, consequently, retains the abilities that `Employees` have. Secretaries also have one additional ability: the ability to take dictation. If we wrote `Secretary` as a standalone class, its code would not reflect this relationship very elegantly. We would be forced to repeat all of the same methods from `Employee` with identical behavior. Here is the redundant class:

```
1   // A redundant class to represent secretaries.
2   public class Secretary {
3       public int getHours() {
4           return 40;
5       }
6
7       public double getSalary() {
8           return 40000.0;
9       }
10
11      public int getVacationDays() {
12          return 10;
13      }
14
15      public String getVacationForm() {
16          return "yellow";
17      }
18
19      // this is the only added behavior
20      public void takeDictation(String text) {
21          System.out.println("Dictating text: " + text);
22      }
23  }
```

The only code unique to the `Secretary` class is its `takeDictation` method. What we'd really like to do is to be able to copy the behavior from class `Employee` without rewriting it in the `Secretary` class file.

```
public class Secretary {
    copy all the methods from the Employee class.

    // this is the only added behavior
    public void takeDictation() {
        System.out.println("I know how to take dictation.");
    }
}
```

Fortunately, Java provides a mechanism called *inheritance* that can help us remove this sort of redundancy between similar classes of objects. Inheritance allows the programmer to specify a relationship between two classes in which one class includes ("inherits") the state and behavior of another.

Inheritance (Inherit)

A programming technique that allows a derived class to extend the functionality of a base class, inheriting all of its state and behavior.

The derived class, more commonly called the *subclass*, inherits all of the state and behavior of its parent class, commonly called the *superclass*.

Superclass

The parent class in an inheritance relationship.

Subclass

The child, or derived, class in an inheritance relationship.

We say that the subclass *extends* the superclass because it not only receives the superclass's state and behavior but can also add new state and behavior of its own. The subclass can also replace inherited behavior with new behavior as needed, which we'll discuss in the next section.

A Java class can have only one superclass; it is not possible to extend more than one class. This is called *single inheritance*. On the other hand, one class may be extended by many subclasses.

To declare one class as the subclass of another, place the `extends` keyword followed by the superclass name at the end of the subclass header. The general syntax is the following:

```
public class <name> extends <superclass> {
    ...
}
```

We can rewrite the `Secretary` class to extend the `Employee` class. This will create an is-a relationship in which every `Secretary` also is an `Employee`. `Secretary`

objects will inherit copies of the `getHours`, `getSalary`, `getVacationDays`, and `getVacationForm` methods, so we won't need to write these methods in the `Secretary` class. This will remove the redundancy between the classes.

It's legal and expected for a subclass to add new behavior that wasn't present in the superclass. We said previously that secretaries add an ability not seen in other employees: the ability to take dictation. We can add this to our otherwise empty `Secretary` class. The following is the complete `Secretary` class:

```
1 // A class to represent secretaries.
2 public class Secretary extends Employee {
3     public void takeDictation(String text) {
4         System.out.println("Dictating text: " + text);
5     }
6 }
```

This concise new version of the `Secretary` class has the same behavior as the longer class shown before. Like the two-page specialized manual, this class shows only the features that are unique to the specific job class. In this case, it is very easy to see that the unique behavior of secretaries in our system is to take dictation.

The following client code would work with our new `Secretary` class:

```
1  public class EmployeeMain {
2      public static void main(String[] args) {
3          System.out.print("Employee: ");
4          Employee edna = new Employee();
5          System.out.print(edna.getHours() + ", ");
6          System.out.printf("$%.2f, ", edna.getSalary());
7          System.out.print(edna.getVacationDays() + ", ");
8          System.out.println(edna.getVacationForm());
9
10         System.out.print("Secretary: ");
11         Secretary stan = new Secretary();
12         System.out.print(stan.getHours() + ", ");
13         System.out.printf("$%.2f, ", stan.getSalary());
14         System.out.print(stan.getVacationDays() + ", ");
15         System.out.println(stan.getVacationForm());
16         stan.takeDictation("hello");
17     }
18 }
```

The code would produce the following output:

```
Employee: 40, $40000.00, 10, yellow
Secretary: 40, $40000.00, 10, yellow
Dictating text: hello
```

Notice that the first four methods produce the same output for both objects, because `Secretary` inherits that behavior from `Employee`. The fifth line of `Secretary` output reflects the new extended behavior of the `takeDictation` method.

Overriding Methods

We can use inheritance in our other types of `Employees`, creating `Lawyer`, `LegalSecretary`, and `Marketer` classes that are subclasses of `Employee`. But while the `Secretary` class merely adds behavior to the standard `Employee` behavior, these other classes also need to replace some of the standard `Employee` behavior with their own. Lawyers receive three weeks of vacation and use a pink form to apply for vacation, and they know how to handle lawsuits. Legal secretaries receive $45,000 a year (a $5,000 raise over the standard amount), they know how to take dictation (like regular secretaries), and they can file legal briefs. Marketers receive $50,000 a year (a $10,000 raise over the standard amount), and they know how to advertise.

We'd like these new classes to inherit most of the behavior from the `Employee` class, but we need to change or replace certain parts of the behavior. It's legal to replace superclass behavior by writing new versions of the relevant method(s) in the subclasses. The new version in the subclass will replace the one inherited from `Employee`. This idea of replacing behavior from the superclass is called *overriding*.

> **Override**
>
> To implement a new version of a method to replace code that would otherwise have been inherited from a superclass.

To override a method, just write the method you want to replace in the subclass. No special syntax is required, but the method's name and signature must exactly match those of the method from the superclass.

Here is the `Lawyer` class that extends `Employee` and overrides the relevant methods:

```
1  // A class to represent lawyers.
2  public class Lawyer extends Employee {
3      // overrides getVacationDays from Employee class
4      public int getVacationDays() {
5          return 15;
6      }
7
8      // overrides getVacationForm from Employee class
9      public String getVacationForm() {
10         return "pink";
11     }
12
```

```
13      // this is the Lawyer's added behavior
14      public void sue() {
15          System.out.println("I'll see you in court!");
16      }
17  }
```

The `LegalSecretary` class could also be written to extend `Employee`, but it has more in common with `Secretary`. It is legal for a class to extend a class that itself extends a class, creating a multi-level hierarchy. So we will write the class for `LegalSecretary` as an extension of the class `Secretary` so it inherits the ability to take dictation:

```
1  // A class to represent legal secretaries.
2  public class LegalSecretary extends Secretary {
3      // overrides getSalary from Employee class
4      public double getSalary() {
5          return 45000.0;
6      }
7
8      // new behavior of LegalSecretary objects
9      public void fileLegalBriefs() {
10         System.out.println("I could file all day!");
11     }
12 }
```

The following client program uses our `Lawyer` and `LegalSecretary` classes. Notice that the legal secretary inherits not only the normal employee behavior, but also the behavior to take dictation from the standard secretary:

```
1  public class EmployeeMain2 {
2      public static void main(String[] args) {
3          System.out.print("Lawyer: ");
4          Lawyer lucy = new Lawyer();
5          System.out.print(lucy.getHours() + ", ");
6          System.out.printf("$%.2f, ", lucy.getSalary());
7          System.out.print(lucy.getVacationDays() + ", ");
8          System.out.println(lucy.getVacationForm());
9          lucy.sue();
10
11         System.out.print("Legal Secretary: ");
12         LegalSecretary leo = new LegalSecretary();
13         System.out.print(leo.getHours() + ", ");
14         System.out.printf("$%.2f, ", leo.getSalary());
15         System.out.print(leo.getVacationDays() + ", ");
```

```
16          System.out.println(leo.getVacationForm());
17          leo.takeDictation("neato");
18          leo.fileLegalBriefs();
19      }
20  }
```

The program produces the following output:

```
Lawyer: 40, $40000.00, 15, pink
I'll see you in court!
Legal Secretary: 40, $45000.00, 10, yellow
Dictating text: neato
I could file all day!
```

Be careful not to confuse overriding with overloading. Overloading, introduced in Chapter 3, occurs when one class contains multiple methods that have the same name but different parameter signatures. Overriding occurs when a subclass substitutes its own version of an otherwise inherited method that uses exactly the same name and the same parameters.

9.2 Interacting with the Superclass


VideoNote

The classes in the previous sections demonstrated inheritance in classes containing only methods. But you'll want to write more meaningful classes that use inheritance with fields, methods, and constructors. These subclasses require more complex interaction with the state and behavior they inherit from their superclass. To show you how to perform this interaction properly, we'll need to introduce a new keyword called `super`.

Calling Overridden Methods

Suppose things are going well at our example legal firm, so the company decides to give every employee a $10,000 raise. What is the best way to enact this policy change in the code we've written? We can edit the `Employee` class and change its `getSalary` method to return 50000 instead of 40000. But some of the other types of employees have salaries higher than the original $40,000 rate (legal secretaries at $45,000 and marketers at $50,000), and these will need to be raised as well. We'll end up needing to change several files to enact this single overall raise.

The problem is that our existing code does not represent the relationship between the various salaries very well. For example, in the `LegalSecretary` class, instead of saying that we want to return 45000, we'd like to return the `Employee` class's salary plus $5000. You may want to write code like the following, but it does not work,

because the `getSalary` method has overridden the one from the superclass, meaning that the code that follows calls itself infinitely:

```
// flawed implementation of LegalSecretary salary code
public double getSalary() {
    return getSalary() + 5000;
}
```

You also can't just call the `Employee` version of `getSalary` by writing `Employee.getSalary()`, because that is the syntax for executing static methods, not instance methods of objects. Instead, Java provides a keyword `super` that refers to a class's superclass. This keyword is used when calling a superclass method or constructor. Here is the general syntax for calling an overridden method using the `super` keyword:

```
super.<method name>(<expression>, <expression>, ..., <expression>)
```

The correct version of the legal secretary's salary code is the following. Writing the marketer's version is left as an exercise.

```
// working LegalSecretary salary code
public double getSalary() {
    return super.getSalary() + 5000; // $5k more than general employees
}
```

Accessing Inherited Fields

To examine the interactions of more complex classes in a hierarchy, let's shift to a more complex example than the employee classes we've been using so far. In Chapter 8's case study, we built a `Stock` class representing purchased shares of a given stock. Here's the code for that class, which has been shortened a bit for this section by removing tests for illegal arguments:

```
1  // A Stock object represents purchases of shares of a stock.
2  public class Stock {
3      private String symbol;
4      private int totalShares;
5      private double totalCost;
6
7      // initializes a new Stock with no shares purchased
8      public Stock(String symbol) {
9          this.symbol = symbol;
10         totalShares = 0;
11         totalCost = 0.0;
12     }
```

```
13
14      // returns the total profit or loss earned on this stock
15      public double getProfit(double currentPrice) {
16          double marketValue = totalShares * currentPrice;
17          return marketValue - totalCost;
18      }
19
20      // records purchase of the given shares at the given price
21      public void purchase(int shares, double pricePerShare) {
22          totalShares += shares;
23          totalCost += shares * pricePerShare;
24      }
25  }
```

Now let's imagine that you want to create a type of object for stocks which pay dividends. Dividends are profit-sharing payments that a corporation pays its shareholders. The amount that each shareholder receives is proportional to the number of shares that person owns. Not every stock pays dividends, so you wouldn't want to add this functionality directly to the `Stock` class. Instead, you should create a new class called `DividendStock` that extends `Stock` and adds this new behavior.

Each `DividendStock` object will inherit the symbol, total shares, and total cost from the `Stock` superclass. You'll simply need to add a field to record the amount of the dividends paid:

```
public class DividendStock extends Stock {
    private double dividends; // amount of dividends paid
    ...
}
```

Using the `dividends` field, you can write a method in the `DividendStock` class that lets the shareholder receive a per-share dividend. Your first thought might be to write code like the following, but this won't compile:

```
// this code does not compile
public void payDividend(double amountPerShare) {
    dividends += amountPerShare * totalShares;
}
```

A `DividendStock` cannot access the `totalShares` field it has inherited, because `totalShares` is declared private in `Stock`. A subclass may not refer directly to any private fields that were declared in its superclass, so you'll get a compiler error like the following:

```
DividendStock.java:17: totalShares has private access in Stock
```

It may seem unnecessarily restrictive that a class isn't able to examine the fields it has inherited, since those fields are part of the object. The reason Java is built this way is to prevent a subclass from violating the encapsulation of the superclass. If a superclass object held sensitive data and subclasses were allowed to access that data directly, they could change it in malicious ways the superclass did not intend.

The solution here is to use accessor or mutator methods associated with our fields to access or change their values. The `Stock` class doesn't have a public accessor method for the `totalShares` field, but you can now add a `getTotalShares` method to the `Stock` class:

```
// returns the total shares purchased of this stock
public int getTotalShares() {
    return totalShares;
}
```

Here is a corrected version of the `payDividend` method that uses the `getTotalShares` method from `Stock`:

```
// records a dividend of the given amount per share
public void payDividend(double amountPerShare) {
    dividends += amountPerShare * getTotalShares();
}
```

The `DividendStock` subclass is allowed to call the public `getTotalShares` method, so the code now behaves properly. If we had a similar situation in which subclasses needed to modify total shares from a stock, the `Stock` class would need to provide a `setTotalShares` method or something similar.

Calling a Superclass's Constructor

Unlike other behaviors, constructors are not inherited. You'll have to write your own constructor for the `DividendStock` class, and when you do so the problem of the inability to access private fields will arise again.

The `DividendStock` constructor should accept the same parameter as the `Stock` constructor: the stock symbol. It should have the same behavior as the `Stock` constructor but should also initialize the `dividends` field to `0.0`. The following constructor implementation might seem like a good start, but it is redundant with `Stock`'s constructor and won't compile successfully:

```
// this constructor does not compile
public DividendStock(String symbol) {
    this.symbol = symbol;
    totalShares = 0;
    totalCost = 0.0;
    dividends = 0.0; // this line is the new code
}
```

The compiler produces four errors: one error for each line that tries to access an inherited private field, and a message about a missing `Stock()` constructor:

```
DividendStock.java:5: cannot find symbol
symbol : constructor Stock()
location: class Stock
public DividendStock(String symbol) {
                                    ^
DividendStock.java:6: symbol has private access in Stock
DividendStock.java:7: totalShares has private access in Stock
DividendStock.java:8: totalCost has private access in Stock
```

The first problem is that even though a `DividendStock` does contain the `symbol`, `totalShares`, and `totalCost` fields by inheritance, it cannot refer to them directly because they were declared private in the `Stock` class.

The second problem—the missing `Stock()` constructor—is a subtle and confusing detail of inheritance. A subclass's constructor must always begin by calling a constructor from the superclass. The reason is that a `DividendStock` object partially consists of a `Stock` object, and you must initialize the state of that `Stock` object first by calling a constructor for it. If you don't do so explicitly, the compiler assumes that `Stock` has a parameterless `Stock()` constructor and tries to initialize the `Stock` data by calling this constructor. Since the `Stock` class doesn't actually have a parameterless constructor, the compiler prints a bizarre error message about a missing `Stock()` constructor. (It's a shame that the error message isn't more informative.)

The solution to this problem is to explicitly call the `Stock` constructor that accepts a `String` symbol as its parameter. Java uses the keyword `super` for a subclass to refer to behavior from its superclass. To call a constructor of a superclass, write the keyword `super`, followed by the constructor's parameter values in parentheses:

```
super(<expression>, <expression>, ..., <expression>);
```

In the case of the `DividendStock` constructor, the following code does the trick. Use the `super` keyword to call the superclass constructor, passing it the same `symbol` value that was passed to the `DividendStock` constructor. This action will initialize the `symbol`, `totalShares`, and `totalCost` fields. Then set the initial dividends to `0.0`:

```
// constructs a new dividend stock with the given symbol
// and no shares purchased
public DividendStock(String symbol) {
    super(symbol); // call Stock constructor
    dividends = 0.0;
}
```

The call to the superclass's constructor using `super` must be the first statement in a subclass's constructor. If you reverse the order of the statements in `DividendStock`'s constructor and set the dividends before you call `super`, you'll get a compiler error like the following:

```
Call to super must be first statement in constructor
    super(symbol); // call Stock constructor
    ^
```

Here's the `DividendStock` class so far. The class isn't complete yet because you haven't yet implemented the behavior to make dividend payments:

```
// A DividendStock object represents a stock purchase that also pays
// dividends.
public class DividendStock extends Stock {
    private double dividends; // amount of dividends paid

    // constructs a new dividend stock with the given symbol
    // and no shares purchased
    public DividendStock(String symbol) {
        super(symbol); // call Stock constructor
        this.dividends = 0.0;
    }
    ...
}
```

`DividendStock` Behavior

To implement dividend payments, we'll begin by writing a method called `payDividend` that accepts a dividend amount per share and adds the proper amount to `DividendStock`'s `dividends` field. The amount per share should be multiplied by the number of shares the shareholder owns:

```
// records a dividend of the given amount per share
public void payDividend(double amountPerShare) {
    dividends += amountPerShare * getTotalShares();
}
```

The dividend payments that are being recorded should be considered to be profit for the stockholder. The overall profit of a `DividendStock` object is equal to the profit from the stock's price plus any dividends. This amount is computed as the market value (number of shares times current price) minus the total cost paid for the shares, plus the amount of dividends paid.

Notice that you don't need to use `super.getTotalShares` in the preceding code. You have to use the `super` keyword only when you are accessing overridden methods or constructors from the superclass. `DividendStock` doesn't override the `getTotalShares` method, so you can call it without the `super` keyword.

Because the profit of a `DividendStock` object is computed differently from that of a regular `Stock` object, you should override the `getProfit` method in the `DividendStock` class to implement this new behavior. An incorrect initial attempt might look like the following code:

```
// this code does not compile
public double getProfit(double currentPrice) {
    double marketValue = totalShares * currentPrice;
    return marketValue - totalCost + dividends;
}
```

The preceding code has two problems. First, you can't refer directly to the various fields that were declared in `Stock`. To get around this problem, you can add accessor methods for each field. The second problem is that the code is redundant: It duplicates much of the functionality from `Stock`'s `getProfit` method, which was shown earlier. The only new behavior is the adding of dividends into the total.

To remove this redundancy, you can have `DividendStock`'s `getProfit` method call `Stock`'s `getProfit` method as part of its computation. However, since the two methods share the same name, you must explicitly tell the compiler that you want to call `Stock`'s version. Again, you do this using the `super` keyword.

Here is the corrected code, which does compile and eliminates the previous redundancy:

```
// returns the total profit or loss earned on this stock,
// including profits made from dividends
public double getProfit(double currentPrice) {
    return super.getProfit(currentPrice) + dividends;
}
```

And here is the code for the completed `DividendStock` class:

```
1  // A DividendStock object represents a stock purchase that also pays
2  // dividends.
3  public class DividendStock extends Stock {
4      private double dividends; // amount of dividends paid
5
6      // constructs a new dividend stock with the given symbol
7      // and no shares purchased
8      public DividendStock(String symbol) {
9          super(symbol); // call Stock constructor
10         dividends = 0.0;
11     }
12
13     // returns the total profit or loss earned on this stock,
```

```
14      // including profits made from dividends
15      public double getProfit(double currentPrice) {
16          return super.getProfit(currentPrice) + dividends;
17      }
18
19      // records a dividend of the given amount per share
20      public void payDividend(double amountPerShare) {
21          dividends += amountPerShare * getTotalShares();
22      }
23  }
```

It's possible to have a deeper inheritance hierarchy with multiple layers of inheritance and overriding. However, the `super` keyword reaches just one level upward to the most recently overridden version of the method. It's not legal to use `super` more than once in a row; you cannot make calls like `super.super.getProfit`. If you need such a solution, you'll have to find a workaround such as using different method names.

The `Object` Class

A class called `Object` serves as the ultimate superclass for all other Java classes, even those that do not declare explicit superclasses in their headers. In fact, classes with headers that do not have `extends` clauses are treated as though their headers say `extends Object` when they are compiled. (You can explicitly write the `extends Object` in a class header, but this is unnecessary and not a common style.)

The `Object` class contains methods that are common to all objects. Table 9.1 summarizes the methods of the `Object` class. Note that some of the methods are not public and therefore cannot be called externally.

In Chapter 8 we mentioned that without a `toString` method in a class, the objects will not print properly. For example, our `Point` class printed `Point@119c082` by

Table 9.1 Methods of the `Object` Class

Method	Description
`clone()`	Creates and returns a copy of the object (not a public method)
`equals(obj)`	Indicates whether the other object is equal to this one
`finalize()`	Called automatically by Java when objects are destroyed (not public)
`getClass()`	Returns information about the type of the object
`hashCode()`	Returns a number associated with the object; used in some collections
`toString()`	Returns the state of the object as a `String`
`notify()`, `notifyAll()`, `wait()`	Advanced methods for multithreaded programming

default before we wrote its `toString` method. This default message was the behavior of the `Object` class's `toString` method, which we inherited in our `Point` class. The `Object` class provides a generic `toString` output that will work for every class: the class name followed by some internal numeric information about the object. When we wrote our own `toString` method, we overrode this default behavior.

It is sometimes useful to refer to the `Object` class in your programs. For example, if you wish to write a method that can accept any object as a parameter, you can declare a parameter of type `Object`:

```
// this method can accept any object as its parameter
public static void myMethod(Object o) {
    ...
}
```

Of course, since your parameter can be anything, you are only allowed to call the methods from the `Object` class on it, such as `toString` or `getClass`. It is also legal to have a method whose return type is `Object`.

The `Object` class is used extensively in the Java class libraries. For example, the `println` method of the `PrintStream` class (the class of which `System.out` is an instance) accepts a parameter of type `Object`, which allows you to print any object to the console.

The `equals` Method

For several chapters now, you have used the == operator to compare for equality. You have seen that this operator does not behave as expected when used on objects, because it is possible to have two distinct objects with equivalent states, such as two `Point` objects with the coordinates (5, 2). This observation is a reminder that an object has an *identity* and is distinct from other objects, even if another object happens to have the same state.

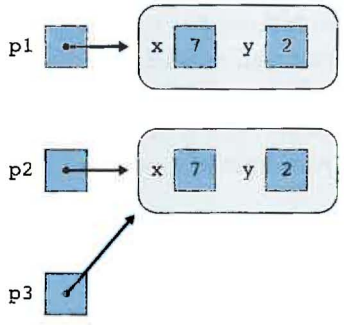
A nonprogramming analogy would be if you and your friend both purchased identical pairs of shoes. They are in some ways equivalent, but you still consider them distinct and separate items. You might not want to share them. And the items' states are not linked in any way. Over time, their state might become visibly unequal, such as if one of you wore your new shoes more often than the other.

The == operator does not behave as expected with objects because it tests whether two objects have the same identity. The == comparison actually tests whether two variables refer to the same object, not whether two distinct objects have the same state. Consider the following three variable declarations:

```
Point p1 = new Point(7, 2);
Point p2 = new Point(7, 2);
Point p3 = p2;
```

The following diagram represents the state of these objects and the variables that refer to them. Notice that `p3` is not a reference to a third object but a second reference

to the object that was referred to by `p2`. This means that a change to `p2`, such as a call of its `translate` method, would also be reflected in `p3`:



In the case of the preceding `Point` objects, the expression `p1 == p2` would evaluate to false because `p1` and `p2` do not refer to the same object. The object referred to by `p1` has the same state as `p2`'s object, but they have different identities. The expression `p2 == p3` would evaluate to `true`, though, because `p2` does refer to the same object as `p3`.

Often, when comparing two objects, we want to know whether the objects have the same state. To perform such a comparison, we use a special method called `equals`. Every Java object contains an `equals` method that it uses to compare itself to other objects.

The previous section mentioned that a class without a `toString` method receives a default version of the method. Similarly, a class without an `equals` method receives a default version that uses the most conservative definition of equality, considering two objects to be equal only if they have the same identity. This means that the default `equals` method behaves identically to the `==` operator. If you want a method that will behave differently, you must write your own `equals` method to replace the default behavior.

A proper `equals` method performs a comparison of two objects' states and returns `true` if the states are the same. With the preceding `Point` objects, we'd like the expressions `p1.equals(p2)`, `p1.equals(p3)`, and `p2.equals(p3)` to evaluate to `true` because the `Points` all have the same (x, y) coordinates.

You can imagine a piece of client code that examines two `Point` objects to see whether they have the same `x` and `y` field values:

```
if (p1.getX() == p2.getX() && p1.getY() == p2.getY()) {
    // the objects have equal state
    ...
}
```

But the `equals` functionality should actually be implemented in the `Point` class itself, not in the client code. Rather than having two `Points`, `p1` and `p2`, the `equals` method considers the first object to be the "implicit parameter" and accepts the second object as a parameter. It returns `true` if the two objects are equal.

The following code is an initial implementation of the `equals` method that has several flaws:

```
// a flawed implementation of an equals method
public boolean equals(Point p2) {
    if (x == p2.getX() && y == p2.getY()) {
        return true;
    } else {
        return false;
    }
}
```

An initial flaw we can correct is that the preceding code doesn't make good use of "Boolean Zen," described in Chapter 5. Recall that when your code uses an `if/else` statement to return a `boolean` value of `true` or `false`, often you can directly return the value of the `if` statement's condition:

```
return x == p2.getX() && y == p2.getY();
```

It's legal for the `equals` method to access `p2`'s fields directly, so we can optionally modify this further. Private fields are visible to their entire class, including other objects of that same class, so it is legal for one `Point` object to examine the fields of another:

```
return x == p2.x && y == p2.y;
```

Some programmers respect `p2`'s encapsulation even against other `Point` objects and therefore would not make the preceding change.

To keep our `equals` method consistent with other Java classes, we must also make a change to its header. The `equals` method's parameter should not be of type `Point`. The method must instead accept a parameter of type `Object`:

```
public boolean equals(Object o)
```

A variable or parameter of type `Object` can refer to any Java object, which means that any object may be passed as the parameter to the `equals` method. Thus, we can compare `Point` objects against any type of object, not just other `Points`. For example, an expression such as `p1.equals("hello")` would now be legal. The `equals` method should return `false` in such a case because the parameter isn't a `Point`.

You might think that the following code would correctly compare the two `Point` objects and return the proper result. Unfortunately, it does not even compile successfully:

```
return x == o.x && y == o.y; // does not compile
```

The Java compiler doesn't allow us to write an expression such as `o.x` because it doesn't know ahead of time whether `o`'s object will have a field called `x`. The

preceding code produces errors such as the following for each of o's fields that we try to access:

```
Point.java:36: cannot find symbol
symbol : variable x
location: class java.lang.Object
```

If we want to treat o as a Point object, we must cast it from type Object to type Point. We've already discussed typecasting to convert between primitive types, such as casting double to int. Casting between object types has a different meaning. A cast of an object is a promise to the compiler. The cast is your assurance that the reference actually refers to a different type and that the compiler can treat it as that type. In our method, we'll write a statement that casts o into a Point object so the compiler will trust that we can access its x and y fields:

```
// returns whether the two Points have the same (x, y) values
public boolean equals(Object o) {
    Point other = (Point) o;
    return x == other.x && y == other.y;
}
```

Don't forget that if your object has fields that are objects themselves, such as a string or Point as a field, then those fields should be compared for equality using their equals method and not using the == operator.

The instanceof Keyword

By changing our equals method's parameter to type Object, we have allowed objects that are not Points to be passed. However, our method still doesn't behave properly when clients pass these objects. An expression in client code such as p.equals("hello") will produce an exception like the following at runtime:

```
Exception in thread "main"
java.lang.ClassCastException: java.lang.String
        at Point.equals(Point.java:25)
        at PointMain.main(PointMain.java:25)
```

The exception occurs because it is illegal to cast a String into a Point; these are not compatible types of objects. To prevent the exception, our equals method will need to examine the type of the parameter and return false if it isn't a Point. The following pseudocode shows the pattern that the code should follow:

```
public boolean equals(Object o) {
    if (o is a Point object) {
        compare the x and y values.
```

```
    } else {
        return false, because o is not a Point object.
    }
}
```

An operator called `instanceof` tests whether a variable refers to an object of a given type. An `instanceof` test is a binary expression that takes the following form and produces a `boolean` result:

```
<expression> instanceof <type>
```

Table 9.2 lists some example expressions using `instanceof` and their results, given the following variables:

```
String s = "carrot";
Point p = new Point(8, 1);
```

Table 9.2 Sample `instanceof` Expressions

Expression	Result
`s instanceof String`	`true`
`s instanceof Point`	`false`
`p instanceof String`	`false`
`p instanceof Point`	`true`
`"hello" instanceof String`	`true`
`null instanceof Point`	`false`

The `instanceof` operator is unusual because it looks like the name of a method but is used more like a relational operator such as > or ==. It is separated from its operands by spaces but doesn't require parentheses, dots, or any other notation. The operand on the left side is generally a variable, and the operand on the right is the name of the class against which you wish to test.

We must examine the parameter `o` in our `equals` method to see whether it is a `Point` object. The following code uses the `instanceof` keyword to implement the `equals` method correctly:

```
// returns whether o refers to a Point with the same (x, y)
// coordinates as this Point
public boolean equals(Object o) {
    if (o instanceof Point) {
        Point other = (Point) o;
        return x == other.x && y == other.y;
    } else { // not a Point object
```

```
            return false;
        }
    }
```

You might think that our `instanceof` test would allow us to remove the type cast below it. After all, the `instanceof` test ensures that the comparison occurs only when `o` refers to a `Point` object. However, the type cast cannot be removed because the compiler doesn't allow the code to compile without it.

A nice side benefit of the `instanceof` operator is that it produces a `false` result when `o` is `null`. Thus, if the client code contains an expression such as `p1.equals(null)`, it will correctly return `false` rather than throwing a `NullPointerException`.

Many classes implement an `equals` method like ours, so much of the preceding `equals` code can be reused as boilerplate code. The following is a template for a well-formed `equals` method. The `instanceof` test and type cast are likely the first two things you'll want to do in any `equals` method that you write:

```
public boolean equals(Object o) {
    if (o instanceof <type>) {
        <type> <name> = (<type>) o;
        <compare the data and return the result.>
    } else {
        return false;
    }
}
```

9.3 Polymorphism


VideoNote

One of the most powerful benefits of inheritance is that it allows client code to treat different kinds of objects in the same way. For example, with the employee class hierarchy described earlier, it's possible for client code to create an array or other data structure that contains both lawyers and legal secretaries, and then perform operations on each element of that array. The client code will behave differently depending on the type of object that is used, because each subclass overrides and changes some of the behavior from the superclass. This ability for the same code to be used with several different types of objects is called *polymorphism.*

> **Polymorphism**
>
> The ability for the same code to be used with several different types of objects and for the code to behave differently depending on the actual type of object used.

Polymorphism is made possible by the fact that the type of a reference variable (one that refers to an object) does not have to exactly match the type of the object it

refers to. More specifically, it is legal for a superclass variable to refer to an object of its subclass. The following is a legal assignment statement:

```
Employee ed = new Lawyer();
```

When we were studying the primitive types, we saw cases in which a variable of one type could store a value of another type (for example, an `int` value can be stored in a `double` variable). In the case of primitive values, Java converts variables from one type to another: `int` values are automatically converted to `double`s when they are assigned.

When a subclass object is stored in a superclass variable, no such conversion occurs. The object referred to by `ed` really is a `Lawyer` object, not an `Employee` object. If we call methods on it, it will behave like a `Lawyer` object. For example, the call of `ed.getVacationForm()` returns `"pink"`, which is the `Lawyer`'s behavior, not the `Employee`'s.

This ability for variables to refer to subclass objects allows us to write flexible code that can interact with many types of objects in the same way. For example, we can write a method that accepts an `Employee` as a parameter, returns an `Employee`, or creates an array of `Employee` objects. In any of these cases, we can substitute a `Secretary`, `Lawyer`, or other subclass object of `Employee`, and the code will still work. Even more importantly, code will actually behave differently depending on which type of object is used, because each subclass overrides and changes some of the behavior from the superclass. This is polymorphism at work.

Here is an example test file that uses `Employee` objects polymorphically as parameters to a static method:

```
1  // Demonstrates polymorphism by passing many types of employees
2  // as parameters to the same method.
3  public class EmployeeMain3 {
4      public static void main(String[] args) {
5          Employee edna = new Employee();
6          Lawyer lucy = new Lawyer();
7          Secretary stan = new Secretary();
8          LegalSecretary leo = new LegalSecretary();
9
10         printInfo(edna);
11         printInfo(lucy);
12         printInfo(stan);
13         printInfo(leo);
14     }
15
16     // Prints information about any kind of employee.
17     public static void printInfo(Employee e) {
18         System.out.print(e.getHours() + ", ");
```

```
19          System.out.printf("$%.2f, ", e.getSalary());
20          System.out.print(e.getVacationDays() + ", ");
21          System.out.print(e.getVacationForm() + ", ");
22          System.out.println(e); // toString representation of employee
23      }
24  }
```

Notice that the method lets us pass many different types of `Employees` as parameters, and it produces different behavior depending on the type that is passed. Polymorphism gives us this flexibility. The last token of output printed for each employee is the employee object itself, which calls the `toString` method on the object. Our classes don't have `toString` methods, so the program uses the default behavior, which prints the class name plus some extra hexadecimal characters. This allows us to distinguish the classes in the output. The program produces output such as the following:

```
40, $40000.00, 10, yellow, Employee@10b30a7
40, $40000.00, 15, pink, Lawyer@1a758cb
40, $40000.00, 10, yellow, Secretary@1b67f74
40, $45000.00, 10, yellow, LegalSecretary@69b332
```

The word "polymorphism" comes from the Greek words "poly" and "morph," which mean "many" and "forms," respectively. The lines of code in the `printInfo` method are polymorphic because their behavior will take many forms depending on what type of employee is passed as the parameter.

The program doesn't know which `getSalary` or `getVacationForm` method to call until it's actually running. When the program reaches a particular call to an object's method, it examines the actual object to see which method to call. This concept has taken many names over the years, such as *late binding*, *virtual binding*, and *dynamic dispatch.*

When you send messages to an object stored in a variable of a superclass type, it is legal only to call methods that are known to the superclass. For example, the following code will not compile because the `Employee` class does not have a `takeDictation` or `fileLegalBriefs` method:

```
Employee ed = new LegalSecretary();
ed.takeDictation("Hello!"); // compiler error
ed.fileLegalBriefs(); // compiler error
```

The compiler does not allow this code because the variable `ed` could theoretically refer to any kind of employee, including one that does not know how to take dictation or file legal briefs. Even though we know it must refer to a `LegalSecretary` because the code is so simple, the compiler enforces this rule strictly and returns an error message. The same thing happens if you write a method that accepts an `Employee` as

a parameter; you cannot call subclass-only methods such as `takeDictation`, `sue`, or `fileLegalBriefs` on the object that is passed in, even if the actual object might have those methods.

The variable's type can be any type equal or higher in the inheritance hierarchy compared to the actual object. So we could store a legal secretary in a variable of type `Secretary`, which would allow us to execute any standard secretary behavior, including taking dictation:

```
Secretary steve = new LegalSecretary();
steve.takeDictation("Hello!"); // OK
steve.fileLegalBriefs(); // compiler error
```

It is legal to cast a variable into a different type of reference in order to make a call on it. This does not change the type of the object, but it promises the compiler that the variable really refers to an object of the other type. For example, the following code works successfully:

```
Employee ed = new Secretary();
((Secretary) ed).takeDictation("Hello!"); // OK
```

You can only cast a reference to a compatible type, one above or below it in its inheritance hierarchy. The preceding code will compile, but it would crash at runtime if the variable `ed` did not actually refer to an object of type `Secretary` or one of its subclasses.

Polymorphism Mechanics

Inheritance and polymorphism introduce some complex new mechanics and behavior into programs. One useful way to get the hang of these mechanics is to perform exercises to interpret the behavior of programs with inheritance. The main goal of these exercises is to help you understand in detail what happens when a Java program with inheritance executes.

The `EmployeeMain3` program developed in the previous section serves as a template for inheritance questions of the following form: Given a certain hierarchy of classes, what behavior would result if we created several objects of the different types and called their various methods on them?

In order to use polymorphism and keep our program compact, we'll store the objects in an array. In the case of the `Employee` hierarchy, it's legal for an object of class `Lawyer`, `Secretary`, or any other subclass of `Employee` to reside as an element of an `Employee[]`.

The following program produces the same output as the `EmployeeMain3` program from the previous section:

```
1  public class EmployeeMain4 {
2      public static void main(String[] args) {
```

```
        Employee[] employees = {new Employee(), new Lawyer(),
            new Secretary(), new LegalSecretary()};

        // print information about each employee
        for (Employee e : employees) {
            System.out.print(e.getHours() + ", ");
            System.out.printf("$%.2f, ", e.getSalary());
            System.out.print(e.getVacationDays() + ", ");
            System.out.print(e.getVacationForm() + ", ");
            System.out.println(e); // calls toString
        }
    }
}
```

Even if you didn't understand inheritance, you might be able to deduce some things about the hierarchy from the classes' names and the relationships among employees in the real world. So let's take this exercise one step further. Instead of using descriptive names for the classes, we'll use letters so that you have to read the code in order to determine the class relationships and behavior.

Assume that the following classes have been defined:

```
public class A {
    public void method1() {
        System.out.println("A 1");
    }

    public void method2() {
        System.out.println("A 2");
    }

    public String toString() {
        return "A";
    }
}
```

```
public class B extends A {
    public void method2() {
        System.out.println("B 2");
    }
}
```

```
public class C extends A {
    public void method1() {
        System.out.println("C 1");
    }

```

```
    public String toString() {
        return "C";
    }
}

public class D extends C {
    public void method2() {
        System.out.println("D 2");
    }
}
```

Consider the following client code that uses these classes. It takes advantage of the fact that every other class extends class `A` (either directly or indirectly), so the array can be of type `A[]`. When you call methods on the elements of the array, polymorphic behavior will result:

```
// Client program to use the A, B, C, and D classes.
public class ABCDMain {
    public static void main(String[] args) {
        A[] elements = {new A(), new B(), new C(), new D()};

        for (int i = 0; i < elements.length; i++) {
            System.out.println(elements[i]);
            elements[i].method1();
            elements[i].method2();
            System.out.println();
        }
    }
}
```

It's difficult to interpret such code and correctly determine its output. In the next section we'll present techniques for doing so.

Interpreting Inheritance Code

To determine the output of a polymorphic program like the one in the previous section, you must determine what happens when each element is printed (i.e., when its `toString` method is called) and when its `method1` and `method2` methods are called. You can draw a diagram of the classes and their methods to see the hierarchy ordering and see which methods exist in each class. Draw each class as a box listing its methods, and connect subclasses to their superclasses with arrows, as shown in Figure 9.2. (This is a simplified version of a common design document called a UML class diagram.)

A good second step is to write a table that lists each class and its methods' output. Write the output not just for the methods defined in each class, but also for the ones that the class inherits. Since the `A` class is at the top of the hierarchy, we'll start there.

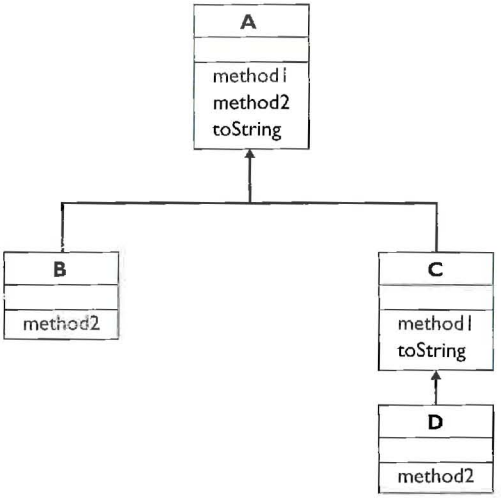


Figure 9.2 Hierarchy of classes A, B, C, and D

When someone calls `method1` on an `A` object, the resulting output is `"A 1"`. When someone calls `method2` on an `A` object, the resulting output is `"A 2"`. When someone prints an `A` object with `toString`, the resulting output is `"A"`. We can fill in the first column of our table as shown in Table 9.3.

Table 9.3 Method Output for Class A

	A
toString	A
method1	A 1
method2	A 2

The next layer in the hierarchy is the `B` class, which inherits all the behavior from `A`, except that it overrides the `method2` output to be `"B 2"`. That means you can fill in the `B` output on your table identically to the `A` output, except that you replace `"A 2"` with `"B 2"`. Table 9.4 shows the table so far.

Table 9.4 Method Output for Classes A and B

	A	B
toString	A	A
method1	A 1	A 1
method2	A 2	B 2

The `C` class also inherits all the behavior from `A`, but it overrides the `method1` output to be `"C 1"` and it overrides the `toString` method to return `"C"`. Thus, the `C` out-

put in the table will have the same second line as the A output, but we replace "A 1" with "C 1" and "A" with "C". Table 9.5 shows the updated table.

Table 9.5 Method Output for Classes A, B, and C

	A	B	C
toString	A	A	C
method1	A 1	A 1	C 1
method2	A 2	B 2	A 2

The D class inherits all the behavior from C, except that it overrides the method2 output to say "D 2". The final output data are shown in Table 9.6.

Table 9.6 Method Output for Classes A, B, C, and D

	A	B	C	D
toString	A	A	C	C
method1	A 1	A 1	C 1	C 1
method2	A 2	B 2	A 2	D 2

Once you've created your table, you can find the output of the client code. The array contains an A object, a B object, a C object, and a D object. For each of these it prints the toString output, then calls method1, then calls method2, then prints a blank line. When a method gets called on an object, you can look up the output of that method for that type in the table. The following is the complete output for the exercise:

```
A
A 1
A 2

A
A 1
B 2

C
C 1
A 2

C
C 1
D 2
```

Interpreting Complex Calls

In a more complicated version of the previous inheritance exercise, a class's methods might call each other or interact with the superclass. Consider the following classes.

Notice that they are listed in random order and that their methods' behavior is more complicated than the previous example:

```
1  public class E extends F {
2      public void method2() {
3          System.out.print("E 2 ");
4          method1();
5      }
6  }
1  public class F extends G {
2      public String toString() {
3          return "F";
5      }
6
7      public void method2() {
8          System.out.print("F 2 ");
9          super.method2();
10     }
11 }
1  public class G {
2      public String toString() {
3          return "G";
5      }
6
7      public void method1() {
8          System.out.print("G 1 ");
9      }
10
11     public void method2() {
12         System.out.print("G 2 ");
13     }
14 }
1  public class H extends E {
2      public void method1() {
3          System.out.print("H 1 ");
5      }
6  }
```

In order to determine the best order in which to examine the classes, you could draw a diagram like the one in Figure 9.2. Then make a table and fill in each class's behavior from top to bottom in the hierarchy. First identify the top class in the hierarchy. Look for the one whose header has no `extends` clause. In this example, that is the `G` class. Its methods have simple behavior, so we can fill in that row of the table immediately, as shown in Table 9.7.

Table 9.7 Method Output for Class G

	G
toString	G
method1	G 1
method2	G 2

The next class in the hierarchy is F, which extends G. The method1 is not overridden, so its output is "G 1 " as it was in the superclass. F does override toString to return "F". It also overrides method2 to print "F 2 " and then call the superclass's (G's) version of method2. When there is a call to a superclass's method, we can evaluate its output immediately and put it into our table by looking at the superclass. This means that the F class's method2 prints "F 2 G 2". Table 9.8 shows this information.

Table 9.8 Method Output for Classes F and G

	F	G
toString	F	G
method1	G 1	G 1
method2	F 2 G 2	G 2

The next class to tackle is E, which extends F. It does not override method1 or toString, so these methods produce the same output as they do in superclass F. Class E does override method2 to print "E 2 " and then call method1. Since method1 prints "G 1 ", calling method2 on an E object prints "E 2 G 1 ".

But here's where things get tricky: You shouldn't write this output in your table. The reason will become clear when we look at the H class, which is a subclass of E that overrides method1. Because of polymorphism, if you call method2 on an H object, when it makes the inner call to method1, it will use the version from the H class. What you should write into your table for E's method2 output is that it prints "E 2 " and then calls method1. Table 9.9 shows the information for class E.

Table 9.9 Method Output for Classes E, F, and G

	E	F	G
toString	F	F	G
method1	G 1	G 1	G 1
method2	E 2 method1()	F 2 G 2	G 2

Lastly we will examine class H, which extends E. It does not override toString, so it produces the same output as in superclass E. It overrides only method1 to print "H 1 ". These methods are simple (they don't call any others), so we can write the toString and method1 output into the table immediately.

Table 9.10 Method Output for Classes E, F, G, and H

	E	F	G	H
toString	F	F	G	F
method1	G 1	G 1	G 1	H 1
method2	E 2 method1()	F 2 G 2	G 2	E 2 method1()

Class H doesn't override method2, so you may think that you can copy over its output from superclass E. But remember that the inherited method2 prints "E 2 " and then calls method1. In this case, we are thinking from the perspective of an H object. So that inner call to method1 now prints "H 1 " instead of the old output. This means that calling method2 on an H object actually prints "E 2 H 1 ". Table 9.10 shows the final output for all methods.

9.4 Inheritance and Design

Inheritance affects the thought processes you should use when designing object-oriented solutions to programming problems. You should be aware of similarities between classes and potentially capture those similarities with inheritance relationships and hierarchies. The designers of the Java class libraries have followed these principles, as we'll see when we examine a graphical subclass in this section.

However, there are also situations in which using inheritance seems like a good choice but turns out to produce poor results. Misuse of inheritance can introduce some pitfalls and problems that we'll now examine.

A Misuse of Inheritance

Imagine that you want to write a program that deals with points in three-dimensional space, such as a three-dimensional game, rendering program, or simulation. A Point3D class would be useful for storing the positions of objects in such a program.

This seems to be a case in which inheritance will be useful to extend the functionality of existing code. Many programmers would be tempted to have Point3D extend Point and simply add the new code for the z-coordinate. Here's a quick implementation of a minimal Point3D class that extends Point:

```
1  // A Point3D object represents an (x, y, z) location.
2  // This is not a good design to follow.
3
4  public class Point3D extends Point {
5      private int z;
6
7      // constructs a new 3D point with the given coordinates
8      public Point3D(int x, int y, int z) {
```

```
 9          super(x, y);
10          this.z = z;
11      }
12
13      // returns the z-coordinate of this Point3D
14      public int getZ() {
15          return z;
16      }
17  }
```

On the surface, this seems to be a reasonable implementation. However, consider the `equals` method defined in the `Point` class. It compares the x- and y-coordinates of two `Point` objects and returns `true` if they are the same:

```
// returns whether o refers to a Point with the same
// (x, y) coordinates as this Point
public boolean equals(Object o) {
    if (o instanceof Point) {
        Point other = (Point) o;
        return x == other.x && y == other.y;
    } else { // not a Point object
        return false;
    }
}
```

You might also want to write an `equals` method for the `Point3D` class. Two `Point3D` objects are equal if they have the same x-, y-, and z-coordinates. The following is a working implementation of `equals` that is correct but stylistically unsatisfactory:

```
public boolean equals(Object o) {
    if (o instanceof Point3D) {
        Point3D p = (Point3D) o;
        return getX() == p.getX() && getY() == p.getY() && z == p.z;
    } else {
        return false;
    }
}
```

The preceding code compiles and runs correctly in many cases, but it has a subtle problem that occurs when you compare `Point` objects to `Point3D` objects. The `Point` class's `equals` method tests whether the parameter is an instance of `Point` and returns `false` if it is not. However, the `instanceof` operator will return `true`

not only if the variable refers to that type, but also if it refers to any of its subclasses. Consider the following test in the `equals` method of the `Point` class:

```
if (o instanceof Point) {
    ...
}
```

The test will evaluate to `true` if `o` refers to a `Point` object or a `Point3D` object. The `instanceof` operator is like an is-a test, asking whether the variable refers to any type that can fill the role of a `Point`. By contrast, `Point3D`'s `equals` method tests whether the parameter is an instance of `Point3D` and rejects it if it is not. A `Point` cannot fill the role of a `Point3D` (not every `Point` is a `Point3D`), so the method will return `false` if the parameter is of type `Point`.

Consequently, the `equals` behavior is not symmetric when it is used with a mixture of `Point` and `Point3D` objects. The following client code demonstrates the problem:

```
Point p = new Point(12, 7);
Point3D p3d = new Point3D(12, 7, 11);
System.out.println("p.equals(p3d) is " + p.equals(p3d));
System.out.println("p3d.equals(p) is " + p3d.equals(p));
```

The code produces the output that follows. The first test returns `true` because a `Point` can accept a `Point3D` as the parameter to `equals`, but the second test returns `false` because a `Point3D` cannot accept a `Point` as its parameter to `equals`:

```
p.equals(p3d) is true
p3d.equals(p) is false
```

This is a problem, because the contract of the `equals` method requires it to be a symmetric operation. You'd encounter other problems if you added more behavior to `Point3D`, such as a `setLocation` or `distance` method.

Proper object-oriented design would not allow `Point3D` to extend `Point`, because any code that asks for a `Point` object should be able to work correctly with a `Point3D` object as well. We call this principle *substitutability*. (It is also sometimes called the Liskov substitution principle, in honor of the Turing Award–winning author of a 1993 paper describing the idea.)

Substitutability

The ability of an object of a subclass to be used successfully anywhere an object of the superclass is expected.

Fundamentally, a `Point3D` isn't the same thing as a `Point`, so an is-a relationship with inheritance is the wrong choice. In this case, you're better off writing `Point3D` from scratch and avoiding these thorny issues.

Is-a Versus Has-a Relationships

There are ways to connect related objects without using inheritance. Consider the task of writing a `Circle` class, in which each `Circle` object is specified by a center point and a radius. It might be tempting to have `Circle` extend `Point` and add the `radius` field. However, this approach is a poor choice because a class is only supposed to capture one abstraction, and a circle simply isn't a point.

A point does make up a fundamental part of the state of each `Circle` object, though. To capture this relationship in the code, you can have each `Circle` object hold a `Point` object in a field to represent its center. One object containing another as state is called a *has-a relationship*.

> **Has-a Relationship**
>
> A connection between two objects where one has a field that refers to the other. The contained object acts as part of the containing object's state.

Has-a relationships are preferred over is-a relationships in cases in which your class cannot or should not substitute for the other class. As a nonprogramming analogy, people occasionally need legal services in their lives, but most of them will choose to *have* a lawyer handle the situation rather than to *be* a lawyer themselves.

The following code presents a potential initial implementation of the `Circle` class:

```
1  // Represents circular shapes.
2  public class Circle {
3      private Point center;
4      private double radius;
5
6      // constructs a new circle with the given radius
7      public Circle(Point center, double radius) {
8          this.center = center;
9          this.radius = radius;
10     }
11
12     // returns the area of this circle
13     public double getArea() {
14         return Math.PI * radius * radius;
15     }
16 }
```

This design presents a `Circle` object as a single clear abstraction and prevents awkward commingling of `Circle` and `Point` objects.

Graphics2D

Use of inheritance is prevalent in the Java class libraries. One notable example is in the drawing of two-dimensional graphics. In this section we'll discuss a class that uses inheritance to draw complex two-dimensional shapes and to assign colors to them.

In the Chapter 3 supplement on graphics, we introduced an object called `Graphics` which acts like a pen that you can use to draw shapes and lines onto a window. When Java's designers wanted additional graphical functionality, they extended the `Graphics` class into a more powerful class called `Graphics2D`. This is a good example of one of the more common uses of inheritance: to extend and reuse functionality from a powerful existing object.

Why didn't the designers of Java simply add the new methods into the existing `Graphics` class? The `Graphics` class already worked properly, so they decided it was best not to perform unnecessary surgery on it. John Vlissides, part of a famous foursome of software engineers affectionately called the "Gang of Four," once described the idea this way: "A hallmark—if not the hallmark—of good object-oriented design is that you can modify and extend a system by adding code rather than by hacking it. In short, change is additive, not invasive."

Making `Graphics2D` extend `Graphics` retains *backward compatibility*. Backward compatibility is the ability of new code to work correctly with old code without modifying the old code. Leaving `Graphics` untouched ensured that old programs would keep working properly and gave new programs the option to use the new `Graphics2D` functionality.

The documentation for `Graphics2D` describes the purpose of the class as follows. "This `Graphics2D` class extends the `Graphics` class to provide more sophisticated control over geometry, coordinate transformations, color management, and text layout. This is the fundamental class for rendering 2-dimensional shapes, text, and images on the Java™ platform." To be specific, `Graphics2D` adds the ability to perform transformations such as scaling and rotation when you're drawing. These capabilities can lead to some fun and interesting images on the screen.

If you used the `DrawingPanel` class from Chapter 3's graphical supplement, you previously wrote statements like the following to get access to the panel's `Graphics` object:

```
Graphics g = panel.getGraphics();
```

Actually, the `getGraphics` method doesn't return a `Graphics` object at all, but rather a `Graphics2D` object. Because of polymorphism, though, it is legal for your program to treat it as a `Graphics` object, because every `Graphics2D` object "is" a `Graphics` object. To use it as a `Graphics2D` object instead, simply write the following line of code:

```
Graphics2D g2 = panel.getGraphics();
```

Table 9.11 Useful Methods of `Graphics2D` Objects

Method	Description
`rotate(angle)`	Rotates subsequently drawn items by the given angle in radians with respect to the origin
`scale(sx, sy)`	Adjusts the size of any subsequently drawn items by the given factors (1.0 means equal size)
`shear(shx, shy)`	Gives a slant to any subsequently drawn items
`translate(dx, dy)`	Shifts the origin by (*dx*, *dy*) in the current coordinate system

Table 9.11 lists some of `Graphics2D`'s extra methods.

The following program demonstrates `Graphics2D`. The `rotate` method's parameter is an angle of rotation measured in radians instead of degrees. Rather than memorizing the conversion between degrees and radians, we can use a static method from the `Math` class called `toRadians` that converts a degree value into the equivalent radian value:

```
1  // Draws a picture of rotating squares using Graphics2D.
2
3  import java.awt.*;
4
5  public class FancyPicture {
6      public static void main(String[] args) {
7          DrawingPanel panel = new DrawingPanel(250, 220);
8          Graphics2D g2 = panel.getGraphics();
9          g2.translate(100, 120);
10         g2.fillRect(-5, -5, 10, 10);
11
12         for (int i = 0; i <= 12; i++) {
13             g2.setColor(Color.BLUE);
14             g2.fillRect(20, 20, 20, 20);
15
16             g2.setColor(Color.BLACK);
17             g2.drawString("" + i, 20, 20);
18
19             g2.rotate(Math.toRadians(30));
20             g2.scale(1.1, 1.1);
21         }
22     }
23 }
```

Figure 9.3 shows the program's output.

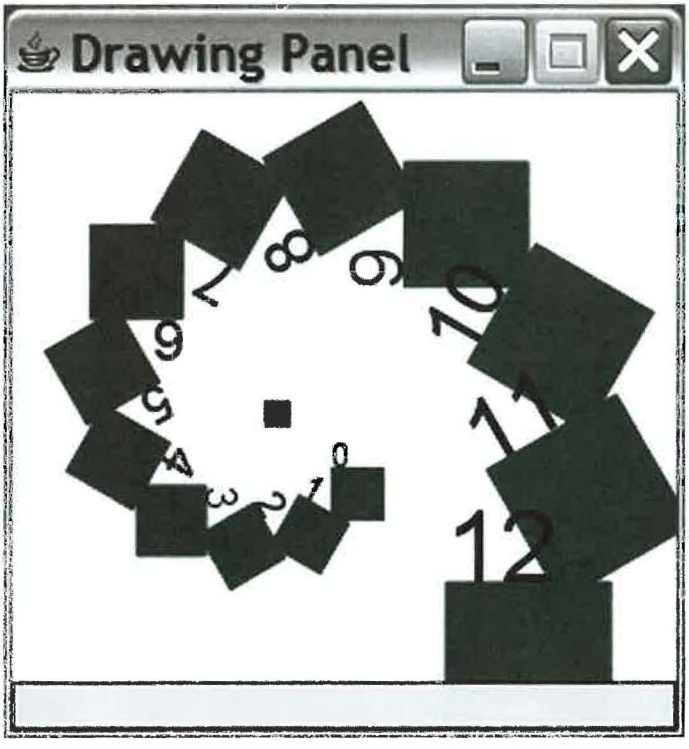


Figure 9.3 Output of `FancyPicture`

9.5 Interfaces

VideoNote

Inheritance is a very useful tool because it enables polymorphism and code sharing, but it does have several limitations. Because Java uses single inheritance, a class can extend only one superclass. This makes it impossible to use inheritance to set up multiple is-a relationships for classes that share multiple characteristics, such as an employee who is both part-time and a secretary. (Some languages, such as C++, do allow multiple inheritance, but this can be complicated and cause subtle problems, so the designers of Java left it out of the language.) There are also situations in which you want is-a relationships and polymorphism but you don't want to share code, in which case inheritance isn't the right tool for the job.

To this end, Java provides a feature called an *interface* that can represent a common supertype between several classes without code sharing.

Interface

A type that consists of a set of method declarations; when classes promise to implement an interface, you can treat those classes similarly in your code.

An interface is like a class, but it contains only method headers without bodies. A class can promise to *implement* an interface, meaning that the class promises to provide implementations of all the methods that are declared in the interface. Classes that implement an interface form an is-a relationship with that interface. In a system with an interface that is implemented by several classes, polymorphic code can handle objects from any of the classes that implement that interface.

A nonprogramming analogy for an interface is a professional certification. It's possible for a person to become certified as a teacher, nurse, accountant, or doctor. To do this, the person must demonstrate certain abilities required of members of those professions. When employers hire someone who has received the proper certification, they expect that the person will be able to perform certain job duties. An interface acts as a certification that classes can meet by implementing all the behavior described in the interface. Code that receives an object implementing an interface can rely on the object having certain behavior.

Interfaces are also used to define roles that objects can play; for example, a `Date` class might implement the `Comparable` interface to indicate that `Date` objects can be compared to each other, or a `Point` class might implement the `Cloneable` interface to indicate that a `Point` object can be replicated.

Interfaces are used in many other places in Java's class libraries. Here are just a few of Java's important interfaces:

- The `ActionListener` interface in the `java.awt` package is used to assign behavior to events when a user clicks on a button or other graphical control.
- The `Serializable` interface in the `java.io` package denotes classes whose objects may be saved to files and transferred over a network.
- The `Comparable` interface allows you to describe how to compare objects of your type to determine which are less than, greater than, or equal to each other. This technique can be used to search or sort a collection of objects.
- The `Formattable` interface lets objects describe different ways that they can be printed by the `System.out.printf` command.
- The `Runnable` interface is used for multithreading, which allows a program to execute two pieces of code at the same time.
- Interfaces such as `List`, `Set`, `Map`, and `Iterator` in the `java.util` package describe data structures that you can use to store collections of objects.

We will cover some of these interfaces in later chapters.

An Interface for Shapes

In this section we'll use an interface to define a polymorphic hierarchy of shape classes without sharing code between them. Imagine that we are creating classes to represent many different types of shapes, such as rectangles, circles, and triangles. We might be tempted to use inheritance with these shape classes because they seem to share some common behavior (all shapes have an area and a perimeter, for example).

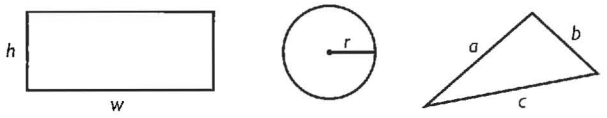


Figure 9.4 Three types of shapes

There is an is-a relationship here, because a rectangle, a circle, and a triangle are all shapes. But code sharing isn't useful in this case because each class implements its behavior differently. As depicted in Figure 9.4 and Table 9.12, each shape computes its area and perimeter in a totally different way. The w and h represent the rectangle's width and height; the r represents the circle's radius; and the a, b, and c represent the lengths of the triangle's three sides.

Since no code is shared among these classes, we should not create a common superclass to represent their is-a relationship. Java uses single inheritance, and we don't want to use up our only potential inheritance relationship here. A better solution would be to write an interface called `Shape` to represent the common functionality of all shapes: the ability to ask for an area and a perimeter. Our various shape classes will implement this interface.

To write an interface, we create a new file with the same name as the interface's name; our `Shape` interface, for example, would be stored in `Shape.java`. We give the interface a header with the keyword `interface` in place of the word `class`:

```
public interface Shape {
    ...
}
```

Inside the interface, we write headers for each method that we want a `Shape` to contain. But instead of writing method bodies with braces, we simply place a semicolon at the end of each header. We don't specify how the methods are implemented. Instead, we're requiring that any class that wants to be considered a shape must implement these methods. In fact, it isn't legal for an interface to contain method bodies; an interface can only contain method headers and class constants.

Table 9.12 Formulas for Area and Perimeter of Each Shape Type

	Rectangle	Circle	Triangle
Area	$w * h$	πr^2	$\sqrt{s(s-a)(s-b)(s-c)}$ where $s = \frac{a+b+c}{2}$
Perimeter	$2(w+h)$	$2\pi r$	$a+b+c$

The following is the complete code for our `Shape` interface. It declares that shapes have methods to compute their areas and perimeters as type `double`:

```
1  // A general interface for shape classes.
2  public interface Shape {
3      public double getArea();
4      public double getPerimeter();
5  }
```

The methods of an interface are sometimes called *abstract methods* because we only declare their names and signatures; we don't specify how they will be implemented.

> **Abstract Method**
>
> A method that is declared (as in an interface) but not implemented. Abstract methods represent the behavior that a class promises to implement when it implements an interface.

Writing the `public` keyword on an interface's method headers is optional. We chose to include the `public` keyword so that the declarations in the interface would match the headers of the method implementations in the classes. The general syntax we'll use for declaring an interface is the following:

```
public interface <name> {
    public <type> <name>(<type> <name>, ..., <type> <name>);
    public <type> <name>(<type> <name>, ..., <type> <name>);
    ...
    public <type> <name>(<type> <name>, ..., <type> <name>);
}
```

Although superficially, classes and interfaces look alike, an interface cannot be *instantiated;* that is, you cannot create objects of its type. In our case, any code trying to create a `new Shape()` would not compile. It is, however, legal to declare variables of type `Shape` that can refer to any object that implements the `Shape` interface, as we'll explore in a moment.

Implementing an Interface

Now that we've written a `Shape` interface, we want to connect the various classes of shapes to it. To connect a class to our interface with an is-a relationship, we must do two things:

1. Declare that the class "implements" the interface.
2. Implement each of the interface's methods in the class.

The general syntax for declaring that a class implements an interface is the following:

```
public class <name> implements <interface> {
    ...
}
```

We must modify the headers of our various shape classes to indicate that they implement all of the methods in the `Shape` interface. The file `Rectangle.java`, for example, should begin like this:

```
public class Rectangle implements Shape {
    ...
}
```

When we claim that our `Rectangle` class implements `Shape`, we are promising that the `Rectangle` class will contain implementations of the `getArea` and `getPerimeter` methods. If a class claims to implement `Shape` but does not have a suitable `getArea` or `getPerimeter` method, it will not compile. For example, if we leave the body of our `Rectangle` class empty and try to compile it, the compiler will give errors like the following:

```
Rectangle.java:2: Rectangle is not abstract and does not override abstract
    method getPerimeter()
public class Rectangle implements Shape {
       ^
1 error
```

In order for the code to compile, we must implement the `getArea` and `getPerimeter` methods in our `Rectangle` class. We'll define a `Rectangle` object by a width and a height. Since the area of a rectangle is equal to its width times its height, we'll implement the `getArea` method by multiplying its fields. We'll then use the perimeter formula `2 * (w + h)` to implement `getPerimeter`. Here is the complete `Rectangle` class that implements the `Shape` interface:

```
1  // Represents rectangular shapes.
2  public class Rectangle implements Shape {
3      private double width;
4      private double height;
5
6      // constructs a new rectangle with the given dimensions
7      public Rectangle(double width, double height) {
8          this.width = width;
9          this.height = height;
10     }
```

```
11
12      // returns the area of this rectangle
13      public double getArea() {
14          return width * height;
15      }
16
17      // returns the perimeter of this rectangle
18      public double getPerimeter() {
19          return 2.0 * (width + height);
20      }
21  }
```

The other classes of shapes are implemented in a similar fashion. We'll define a `Circle` object to have a field called `radius`. (We'll abandon the center `Point` object used in the `Circle` class earlier in the chapter, since we don't need it here.) We can determine the area by multiplying π by the radius squared. To find the perimeter, we'll use the equation `2 * π * r`. Notice that there is no common code between `Circle` and `Rectangle`, so inheritance is unnecessary. Here is the complete `Circle` class:

```
1  // Represents circular shapes.
2  public class Circle implements Shape {
3      private double radius;
4
5      // constructs a new circle with the given radius
6      public Circle(double radius) {
7          this.radius = radius;
8      }
9
10     // returns the area of this circle
11     public double getArea() {
12         return Math.PI * radius * radius;
13     }
14
15     // returns the perimeter of this circle
16     public double getPerimeter() {
17         return 2.0 * Math.PI * radius;
18     }
19 }
```

Finally, we'll specify a triangle's shape by its three side lengths, a, b, and c. The perimeter of the triangle is simply the sum of the three side lengths. The `getArea` method is a bit trickier, but there is a useful geometric formula called

Heron's formula which says that the area of a triangle with sides of lengths a, b, and c is related to a value s that is equal to half the triangle's perimeter:

$$\text{area} = \sqrt{s(s-a)(s-b)(s-c)} \quad \text{where} \quad s = \frac{a+b+c}{2}$$

Here is a complete version of the `Triangle` class:

```
1  // Represents triangular shapes.
2  public class Triangle implements Shape {
3      private double a;
4      private double b;
5      private double c;
6
7      // constructs a new triangle with the given side lengths
8      public Triangle(double a, double b, double c) {
9          this.a = a;
10         this.b = b;
11         this.c = c;
12     }
13
14     // returns this triangle's area using Heron's formula
15     public double getArea() {
16         double s = (a + b + c) / 2.0;
17         return Math.sqrt(s * (s - a) * (s - b) * (s - c));
18     }
19
20     // returns the perimeter of this triangle
21     public double getPerimeter() {
22         return a + b + c;
23     }
24 }
```

Using these shape classes and their common interface, a client program can now construct an array of shapes, write a method that accepts a general shape as a parameter, and otherwise take advantage of polymorphism.

Benefits of Interfaces

Classes that implement a common interface form a type hierarchy similar to those created by inheritance. The interface serves as a parent type for the classes that implement it. The following diagram represents our type hierarchy after our modifications. The way we represent an interface is similar to the way we represent a class, but we use the word "interface" for clarity. The methods are italicized to emphasize that they are abstract. Figure 9.5 shows our use of dashed lines to connect the classes and the interface they implement.

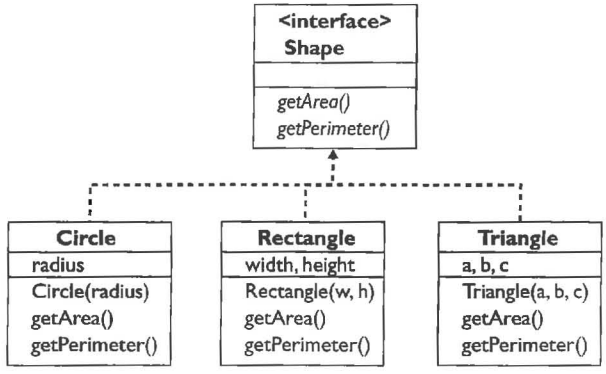


Figure 9.5 Hierarchy of shape classes

The major benefit of interfaces is that we can use them to achieve polymorphism. We can create an array of `Shapes`, pass a `Shape` as a parameter to a method, return a `Shape` from a method, and so on. The following program uses the shape classes in an example that is similar to the polymorphism exercises in Section 9.3:

```
1  // Demonstrates shape classes.
2  public class ShapesMain {
3      public static void main(String[] args) {
4          Shape[] shapes = new Shape[3];
5          shapes[0] = new Rectangle(18, 18);
6          shapes[1] = new Triangle(30, 30, 30);
7          shapes[2] = new Circle(12);
8
9          for (int i = 0; i < shapes.length; i++) {
10             System.out.println("area = " + shapes[i].getArea() +
11                                ", perimeter = " +
12                                shapes[i].getPerimeter());
13         }
14     }
15 }
```

This program produces the following output:

```
area = 324.0, perimeter = 72.0
area = 389.7114317029974, perimeter = 90.0
area = 452.3893421169302, perimeter = 75.39822368615503
```

It would be fairly easy to modify our client program if another shape class, such as `Hexagon` or `Ellipse`, were added to the hierarchy. This is another example of the

desired property of "additive, not invasive" change that we mentioned earlier in this chapter.

It may seem odd that we can have interface variables, arrays, and parameters when it isn't possible to construct an object of an interface type, but all this capacity means is that any object of a type which implements that interface may be used. In our case, any type that implements `Shape` (such as `Circle`, `Rectangle`, or `Triangle`) may be used.

Also recall that interfaces help us cope with the limitations of single inheritance. A class may extend only one superclass but may implement arbitrarily many interfaces. The following is the general syntax for headers of classes that extend a superclass and implement one or more interfaces:

```
public class <name> extends <superclass>
         implements <interface>, <interface>, ..., <interface> {
    ...
}
```

Many classes in the Java class libraries both extend a superclass and implement one or more interfaces. For example, the `PrintStream` class (of which `System.out` is an instance) has the following header:

```
public class PrintStream extends FilterOutputStream
        implements Appendable, Closeable
```

9.6 Case Study: Financial Class Hierarchy

As you write larger and more complex programs, you will end up with more classes and more opportunities to use inheritance and interfaces. It is important to practice devising sensible hierarchies of types, so that you will be able to solve large problems by breaking them down into good classes in the future.

When you are designing an object-oriented system, you should ask yourself the following questions:

- What classes of objects should I write?
- What behavior does the client want each of these objects to have?
- What data do the objects need to store in order to implement this behavior?
- Are the classes related? If so, what is the nature of the relationships?

Having good answers to these questions, along with a good knowledge of the necessary Java syntax, is a good start toward designing an object-oriented system. Such a process is called *object-oriented design*. We discussed some object-oriented design heuristics in the case study of Chapter 8.

Object-Oriented Design (OOD)

Modeling a program or system as a collection of cooperating objects, implemented as a set of classes using class hierarchies.

Let's consider the problem of gathering information about a person's financial investments. We've already explored a `Stock` example in this chapter and the previous chapter, as well as a `DividendStock` class to handle stocks that pay dividends. But stocks are not the only type of asset that investors might have in their financial portfolios. Other investments might include mutual funds, real estate, or cash.

How would you design a complete portfolio system? What new types of objects would you write? Take a moment to consider the problem. We'll discuss an example design next.

Designing the Classes

Each type of asset deserves its own class. We already have the `Stock` class from the last chapter and its `DividendStock` subclass from earlier in this chapter, and we can add classes like `MutualFund` and `Cash`. Each object of each of these types will represent a single investment of that type. For example, a `MutualFund` object will represent a purchase of a mutual fund, and a `Cash` object will represent a sum of money in the user's portfolio. The available types are shown in Figure 9.6. What data and behavior are necessary in each of these types of objects? Take a moment to consider it.

Though each type of asset is unique, the types do have some common behavior: Each asset should be able to compute its current market value and profit or loss, if any. These values are computed in different ways for different asset types, though. For instance, a stock's market value is the total number of shares that the shareholder owns, times the current price per share, while cash is always worth exactly its own amount.

In terms of data, we decided previously that a `Stock` object should store the stock's symbol, the number of shares purchased, the total cost paid for all shares, and the current price of the stock. Dividend stocks also need to store the amount of dividends paid. A `MutualFund` object should store the same data as a `Stock` object, but mutual funds can hold partial shares. Cash only needs to store its own amount. Figure 9.7 updates our diagram of types to reflect this data and behavior.

The asset types are clearly related. Perhaps we'd want to gather and store a person's portfolio of assets in an array. It would be convenient to be able to treat any asset the same way, insofar as they share similar functionality. For example, every asset

Figure 9.6 Financial classes

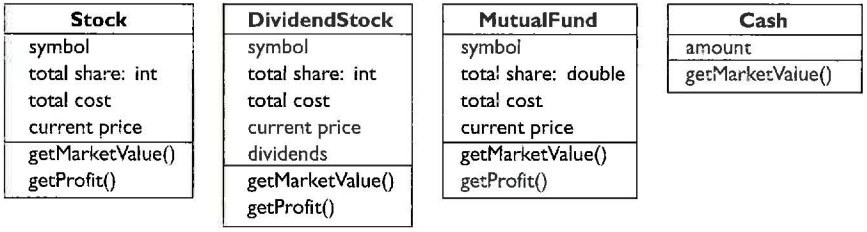


Figure 9.7 Financial classes with state and behavior

has a market value, so it would be nice to be able to compute the total market value of all assets in an investor's portfolio, without worrying about the different types of assets.

Because different assets compute their market values in different ways, we should consider using an interface to represent the notion of an asset and have every class implement the asset interface. Our interface will demand that all assets have methods to calculate the market value and profit. The interface is a way of saying, "Classes that want to consider themselves assets must have `getMarketValue` and `getProfit` methods." Our interface for financial assets would be saved in a file called `Asset.java` and would look like this:

```
1  // Represents financial assets that investors hold.
2  public interface Asset {
3      // how much the asset is worth
4      public double getMarketValue();
5
6      // how much money has been made on this asset
7      public double getProfit();
8  }
```

We'll have our various classes certify that they are assets by making them implement the `Asset` interface. For example, let's look at the `Cash` class. We didn't write a `getProfit` method in our previous diagram of `Cash`, because the value of cash doesn't change and therefore it doesn't have a profit. To indicate this quality, we can write a `getProfit` method for `Cash` that returns `0.0`. The `Cash` class should look like this:

```
1  // A Cash object represents an amount of money held by an investor.
2  public class Cash implements Asset {
3      private double amount; // amount of money held
4
5      // constructs a cash investment of the given amount
6      public Cash(double amount) {
7          this.amount = amount;
```

```
8       }
9
10      // returns this cash investment's market value, which
11      // is equal to the amount of cash
12      public double getMarketValue() {
13          return amount;
14      }
15
16      // since cash is a fixed asset, it never has any profit
17      public double getProfit() {
18          return 0.0;
19      }
20
21      // sets the amount of cash invested to the given value
22      public void setAmount(double amount) {
23          this.amount = amount;
24      }
25  }
```

As we discussed earlier in this chapter, a `DividendStock` is very similar to a normal `Stock`, but it has a small amount of behavior added. Let's display `DividendStock` as a subclass of `Stock` through inheritance, matching the design from earlier in the chapter. Figure 9.8 shows how our hierarchy should now look.

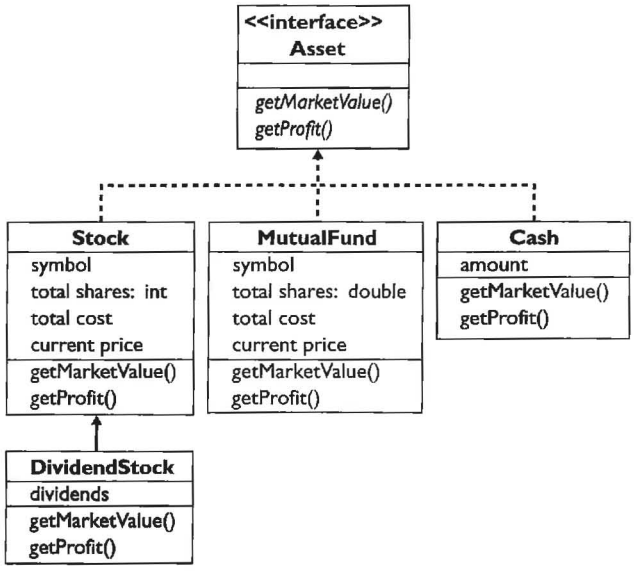


Figure 9.8 Financial class hierarchy

What about the similarity between mutual funds and stocks? They both store assets that are based on shares, with a symbol, total cost, and current price. It wouldn't work very well to make one of them a subclass of the other, though, because the type of shares (integer or real number) isn't the same, and also because it's not a sensible is-a relationship: stocks aren't really mutual funds, and vice versa. It might seem excessive to have a separate class for mutual funds when the only difference is the existence of partial shares. But conceptually, these are separate types of investments, and many investors want to keep them separate. Also, there are some aspects of mutual funds, such as tax ramifications and Morningstar ratings, that are unique and that we might want to add to the program later.

Let's modify our design by making a new superclass called `ShareAsset` which represents any asset that has shares and that contains the common behavior of `Stock` and `MutualFund`. Then we can have both `Stock` and `MutualFund` extend `ShareAsset`, to reduce redundancy.

Our previous versions of the `Stock` and `DividendStock` classes each had a `getProfit` method that required a parameter for the current price per share. In order to implement the `Asset` interface with its parameterless `getMarketValue` and `getProfit` methods, we'll change our design and create a field for the current price. We'll also add methods to get and set the value of the current price. The updated type hierarchy is shown in Figure 9.9.

This practice of redesigning code to meet new requirements is sometimes called *refactoring*.

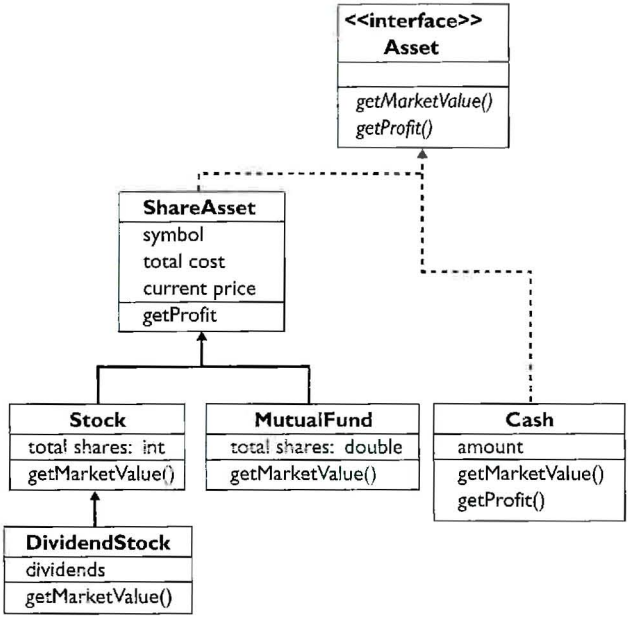


Figure 9.9 Updated financial class hierarchy

Refactoring

Changing a program's internal structure without modifying its external behavior to improve simplicity, readability, maintainability, extensibility, performance, etc.

Redundant Implementation

Here's some potential code for the `ShareAsset` class:

```
1  // A ShareAsset object represents a general asset that has a symbol
2  // and holds shares. Initial version.
3  public class ShareAsset {
4      private String symbol;
5      private double totalCost;
6      private double currentPrice;
7
8      // constructs a new share asset with the given symbol
9      // and current price
10     public ShareAsset(String symbol, double currentPrice) {
11         this.symbol = symbol;
12         this.currentPrice = currentPrice;
13         totalCost = 0.0;
14     }
15
16     // adds a cost of the given amount to this asset
17     public void addCost(double cost) {
18         totalCost += cost;
19     }
20
21     // returns the price per share of this asset
22     public double getCurrentPrice() {
23         return currentPrice;
24     }
25
26     // returns this asset's total cost for all shares
27     public double getTotalCost() {
28         return totalCost;
29     }
30
31     // sets the current share price of this asset
32     public void setCurrentPrice(double currentPrice) {
33         this.currentPrice = currentPrice;
34     }
35 }
```

We stole some code from the `Stock` class and then made a few changes so that the code would fit this interface. Our `Stock` code accepted the current share price as a parameter to its `getProfit` method. Since the `getProfit` method cannot accept any parameters if we wish to implement the interface, we'll instead store the current share price as a field in the `ShareAsset` class and supply a `setCurrentPrice` mutator method that can be called to set its proper value. We also include a constructor that can initialize a `Stock` object with any number of shares and a total cost.

One last modification we made in creating `ShareAsset` was to include an `addCost` method, which we'll use to add a given amount to the asset's total cost. We will need this because purchases of `Stocks` and `MutualFunds` need to update the `totalCost` field, but they cannot do so directly because it is private.

The `Stock` class can now extend `ShareAsset` to implement its remaining functionality. Notice that we both extend `ShareAsset` and implement the `Asset` interface in the class's header:

```
1  // A Stock object represents purchases of shares of a stock.
2  // Initial version.
3  public class Stock extends ShareAsset implements Asset {
4      private int totalShares;
5
6      // constructs a new Stock with the given symbol and
7      // current price per share
8      public Stock(String symbol, double currentPrice) {
9          super(symbol, currentPrice);
10         totalShares = 0;
11     }
12
13     // returns the market value of this stock, which is
14     // the number of total shares times the share price
15     public double getMarketValue() {
16         return totalShares * getCurrentPrice();
17     }
18
19     // returns the total number of shares purchased
20     public int getTotalShares() {
21         return totalShares;
22     }
23
24     // returns the profit made on this stock
25     public double getProfit() {
26         return getMarketValue() - getTotalCost();
27     }
28
29     // records a purchase of the given number of shares of
```

```
30      // the stock at the given price per share
31      public void purchase(int shares, double pricePerShare) {
32          totalShares += shares;
33          addCost(shares * pricePerShare);
34      }
35  }
```

The `MutualFund` class receives similar treatment, but with a `double` for its total shares (the two classes are highly redundant; we'll improve them in the next section):

```
1   // A MutualFund object represents a mutual fund asset.
2   // Initial version.
3   public class MutualFund extends ShareAsset implements Asset {
4       private double totalShares;
5
6       // constructs a new MutualFund investment with the given
7       // symbol and price per share
8       public MutualFund(String symbol, double currentPrice) {
9           super(symbol, currentPrice);
10          totalShares = 0.0;
11      }
12
13      // returns the market value of this mutual fund, which
14      // is the number of shares times the price per share
15      public double getMarketValue() {
16          return totalShares * getCurrentPrice();
17      }
18
19      // returns the number of shares of this mutual fund
20      public double getTotalShares() {
21          return totalShares;
22      }
23
24      // returns the profit made on this mutual fund
25      public double getProfit() {
26          return getMarketValue() - getTotalCost();
27      }
28
29      // records purchase of the given shares at the given price
30      public void purchase(double shares, double pricePerShare) {
31          totalShares += shares;
32          addCost(shares * pricePerShare);
33      }
34  }
```

The `DividendStock` simply adds an amount of dividend payments to a normal `Stock`, which affects its market value. We don't need to override the `getProfit` method in `DividendStock`, because `DividendStock` already inherits a `getProfit` method with the following body:

```
return getMarketValue() - getTotalCost();
```

Notice that `getProfit`'s body calls `getMarketValue`. The `DividendStock` class overrides the `getMarketValue` method, with the convenient side effect that any other method that calls `getMarketValue` (such as `getProfit`) will also behave differently. This occurs because of polymorphism; since `getMarketValue` is overridden, `getProfit` calls the new version of the method. The profit will be correctly computed with dividends because the dividends are added to the market value.

The following code implements the `DividendStock` class:

```
1  // A DividendStock object represents a stock purchase that also pays
2  // dividends.
3  public class DividendStock extends Stock {
4      private double dividends; // amount of dividends paid
5
6      // constructs a new DividendStock with the given symbol
7      // and no shares purchased
8      public DividendStock(String symbol, double currentPrice) {
9          super(symbol, currentPrice); // call Stock constructor
10         dividends = 0.0;
11     }
12
13     // returns this DividendStock's market value, which is
14     // a normal stock's market value plus any dividends
15     public double getMarketValue() {
16         return super.getMarketValue() + dividends;
17     }
18
19     // records a dividend of the given amount per share
20     public void payDividend(double amountPerShare) {
21         dividends += amountPerShare * getTotalShares();
22     }
23 }
```

Abstract Classes

So far we have written classes, which are concrete implementations of state and behavior, and interfaces, which are completely abstract declarations of behavior. There is an entity that exists between these two extremes, allowing us to define some

concrete state and behavior while leaving some abstract, without defined method bodies. Such an entity is called an *abstract class.*

> **Abstract Class**
>
> A Java class that cannot be instantiated, but that instead serves as a superclass to hold common code and declare abstract behavior.

You probably noticed a lot of redundancy between the `Stock` and `MutualFund` code in the last section. For example, although the `getMarketValue` and `getProfit` methods have identical code, they can't be moved up into the `ShareAsset` superclass because they depend on the number of shares, which is different in each child class. Ideally, we should get rid of this redundancy somehow.

There is also a problem with our current `ShareAsset` class. A `ShareAsset` isn't really a type of asset that a person can buy; it's just a concept that happens to be represented in our code. It would be undesirable for a person to actually try to construct a `ShareAsset` object—we wrote the class to eliminate redundancy, not for clients to instantiate it.

We can resolve these issues by designating the `ShareAsset` class as abstract. Writing abstract in a class's header will modify the class in two ways. First, the class becomes noninstantiable, so client code will not be allowed to construct an object of that type with the new keyword. Second, the class is enabled to declare abstract methods without bodies. Unlike an interface, though, an abstract class can also declare fields and implement methods with bodies, so the `ShareAsset` class can retain its existing code.

The general syntax for declaring an abstract class is

```
public abstract class <name> {
    ...
}
```

Thus, our new `ShareAsset` class header will be

```
public abstract class ShareAsset {
    ...
}
```

An attempt to create a `ShareAsset` object will now produce a compiler error such as the following:

```
ShareAsset is abstract; cannot be instantiated
    ShareAsset asset = new ShareAsset("MSFT", 27.46);
    ^
1 error
```

Really, the `Employee` class introduced earlier in this chapter should also have been an abstract class. We did not especially want client code to construct `Employee` objects. No one is *just* an employee; the `Employee` class merely represented a general category that we wanted the other classes to extend.

Abstract classes are allowed to implement interfaces. Rather than requiring all subclasses of `ShareAsset` to implement the `Asset` interface, we can specify that `ShareAsset` implements `Asset`:

```
public abstract class ShareAsset implements Asset {
    ...
}
```

This indication will save `ShareAsset` subclasses from having to write `implements Asset` in their class headers.

`ShareAsset` does not implement the `getMarketValue` method required by `Asset`; that functionality is left for its subclasses. We can instead declare `getMarketValue` as an abstract method in the `ShareAsset` class. Abstract methods declared in abstract classes need to have the keyword `abstract` in their headers in order to compile properly. Otherwise, the syntax is the same as when we declare an abstract method in an interface, with a semicolon replacing the method's body:

```
// returns the current market value of this asset
public abstract double getMarketValue();
```

The general syntax for an abstract method declaration in an abstract class is the following:

```
public abstract <type> <name> (<type> <name>, ..., <type> <name>);
```

Another benefit of this design is that code in the abstract class can actually call any of its abstract methods, even if they don't have implementations in that class. This is allowed because the abstract class can count on its subclasses to implement the abstract methods. Now that `ShareAsset` implements `Asset`, we can move the common redundant `getProfit` code up to `ShareAsset` and out of `Stock` and `MutualFund`:

```
// returns the profit earned on shares of this asset
public double getProfit() {
    // calls an abstract getMarketValue method
    // (the subclass will provide its implementation)
    return getMarketValue() - totalCost;
}
```

`ShareAsset` objects can call `getMarketValue` from their `getProfit` methods even though that method isn't present in `ShareAsset`. The code compiles because

the compiler knows that whatever class extends `ShareAsset` will have to implement `getMarketValue`.

The following is the final version of the `ShareAsset` abstract class:

```
1  // A ShareAsset represents a general asset that has a symbol and
2  // holds shares.
3  public abstract class ShareAsset implements Asset {
4      private String symbol;
5      private double totalCost;
6      private double currentPrice;
7
8      // constructs a new share asset with the given symbol
9      // and current price
10     public ShareAsset(String symbol, double currentPrice) {
11         this.symbol = symbol;
12         this.currentPrice = currentPrice;
13         totalCost = 0.0;
14     }
15
16     // adds a cost of the given amount to this asset
17     public void addCost(double cost) {
18         totalCost += cost;
19     }
20
21     // returns the price per share of this asset
22     public double getCurrentPrice() {
23         return currentPrice;
24     }
25
26     // returns the current market value of this asset
27     public abstract double getMarketValue();
28
29     // returns the profit earned on shares of this asset
30     public double getProfit() {
31         // calls an abstract getMarketValue method
32         // (the subclass will provide its implementation)
33         return getMarketValue() - totalCost;
34     }
35
36     // returns this asset's total cost for all shares
37     public double getTotalCost() {
38         return totalCost;
39     }
```

```
40
41      // sets the current share price of this asset
42      public void setCurrentPrice(double currentPrice) {
43          this.currentPrice = currentPrice;
44      }
45  }
```

An abstract class is a useful hybrid that can contain both abstract and nonabstract methods. All methods declared in an interface are implicitly abstract; they can be declared with the abstract keyword if you wish. Declaring them without the abstract keyword as we have done in this chapter is a commonly used shorthand for the longer explicit form. Unfortunately, abstract classes disallow this shorthand to avoid ambiguity.

Nonabstract classes like `Stock` and `MutualFund` are sometimes called *concrete classes* to differentiate them from abstract classes. We can modify the `Stock` and `MutualFund` classes to take advantage of `ShareAsset` and reduce redundancy. The following are the final versions of the `Stock` and `MutualFund` classes. (`DividendStock` is unmodified.) Notice that the subclasses of `ShareAsset` must implement `getMarketValue`, or we'll receive a compiler error:

```
1   // A Stock object represents purchases of shares of a stock.
2   public class Stock extends ShareAsset {
3       private int totalShares;
4
5       // constructs a new Stock with the given symbol and
6       // current price per share
7       public Stock(String symbol, double currentPrice) {
8           super(symbol, currentPrice);
9           totalShares = 0;
10      }
11
12      // returns the market value of this stock, which is
13      // the number of total shares times the share price
14      public double getMarketValue() {
15          return totalShares * getCurrentPrice();
16      }
17
18      // returns the total number of shares purchased
19      public int getTotalShares() {
20          return totalShares;
21      }
22
23      // records a purchase of the given number of shares of
24      // the stock at the given price per share
```

```
25      public void purchase(int shares, double pricePerShare) {
26          totalShares += shares;
27          addCost(shares * pricePerShare);
28      }
29  }
```

```
1   // A MutualFund object represents a mutual fund asset.
2   public class MutualFund extends ShareAsset {
3       private double totalShares;
4
5       // constructs a new MutualFund investment with the given
6       // symbol and price per share
7       public MutualFund(String symbol, double currentPrice) {
8           super(symbol, currentPrice);
9           totalShares = 0.0;
10      }
11
12      // returns the market value of this mutual fund, which
13      // is the number of shares times the price per share
14      public double getMarketValue() {
15          return totalShares * getCurrentPrice();
16      }
17
18      // returns the number of shares of this mutual fund
19      public double getTotalShares() {
20          return totalShares;
21      }
22
23      // records purchase of the given shares at the given price
24      public void purchase(double shares, double pricePerShare) {
25          totalShares += shares;
26          addCost(shares * pricePerShare);
27      }
28  }
```

Abstract classes can do everything interfaces can do and more, but this does not mean that it is always better to use them than interfaces. One important difference between interfaces and abstract classes is that a class may choose to implement arbitrarily many interfaces, but it can extend just one abstract class. That is why an interface often forms the top of an inheritance hierarchy, as our `Asset` interface did in this design. Such placement allows classes to become part of the hierarchy without having it consume their only inheritance relationships.

Chapter Summary

Inheritance is a feature of Java programs that allows the creation of a parent–child relationship between two types.

The child class of an inheritance relationship (commonly called a subclass) will receive a copy of ("inherit") every field and method from the parent class (superclass). The subclass "extends" the superclass, because it can add new fields and methods to the ones it inherits from the superclass.

A subclass can override a method from the superclass by writing its own version, which will replace the one that was inherited.

Treating objects of different types interchangeably is called polymorphism.

Subclasses can refer to the superclass's constructors or methods using the `super` keyword.

The `Object` class represents the common superclass of all objects and contains behavior that every object should have, such as the `equals` and `toString` methods.

Inheritance provides an "is-a" relationship between two classes. If the two classes are not closely related, inheritance may be a poor design choice and a "has-a" relationship between them (in which one object contains the other as a field) may be better.

An interface is a list of method declarations. An interface specifies method names, parameters, and return types but does not include the bodies of the methods. A class can implement (i.e., promise to implement all of the methods of) an interface.

Interfaces help us achieve polymorphism so that we can treat several different classes in the same way. If two or more classes both implement the same interface, we can use either of them interchangeably and can call any of the interface's methods on them.

An abstract class cannot be instantiated. No objects of the abstract type can be constructed. An abstract class is useful because it can be used as a superclass and can also define abstract behavior for its subclasses to implement.

An abstract class can contain abstract methods, which are declared but do not have bodies. All subclasses of an abstract class must implement the abstract superclass's abstract methods.

Self-Check Problems

Section 9.1: Inheritance Basics

1. What is code reuse? How does inheritance help achieve code reuse?
2. What is the difference between overloading and overriding a method?
3. Which of the following is the correct syntax to indicate that class `A` is a subclass of `B`?

   a. `public class B extends A {`
   b. `public class A : super B {`
   c. `public A(super B) {`
   d. `public class A extends B {`
   e. `public A implements B {`

4. Consider the following classes:

```
public class Vehicle {...}

public class Car extends Vehicle {...}

public class SUV extends Car {...}
```

Which of the following are legal statements?

a. `Vehicle v = new Car();`
b. `Vehicle v = new SUV();`
c. `Car c = new SUV();`
d. `SUV s = new SUV();`
e. `SUV s = new Car();`
f. `Car c = new Vehicle();`

Section 9.2: Interacting with the Superclass

5. Explain the difference between the `this` keyword and the `super` keyword. When should each be used?

6. For the next three problems, consider the following class:

```
1  // Represents a university student.
2  public class Student {
3      private String name;
4      private int age;
5
6      public Student(String name, int age) {
7          this.name = name;
8          this.age = age;
9      }
10
11     public void setAge(int age) {
12         this.age = age;
13     }
14 }
```

Also consider the following partial implementation of a subclass of `Student` to represent undergraduate students at a university:

```
public class UndergraduateStudent extends Student {
    private int year;
    ...
}
```

Can the code in the `UndergraduateStudent` class access the `name` and `age` fields it inherits from `Student`? Can it call the `setAge` method?

7. Write a constructor for the `UndergraduateStudent` class that accepts a name as a parameter and initializes the `UnderGraduateStudent`'s state with that name, an `age` value of `18`, and a `year` value of `0`.

8. Write a version of the `setAge` method in the `UndergraduateStudent` class that not only sets the `age` but also increments the `year` field's value by one.

9. Consider the following two automobile classes:

```
public class Car {
    public void m1() {
        System.out.println("car 1");
    }

    public void m2() {
        System.out.println("car 2");
    }

    public String toString() {
        return "vroom";
    }
}
public class Truck extends Car {
    public void m1() {
        System.out.println("truck 1");
    }
}
```

Given the following declared variables, what is the output from the following statements?

```
Car mycar = new Car();
Truck mytruck = new Truck();

System.out.println(mycar);
mycar.m1();
mycar.m2();
System.out.println(mytruck);
mytruck.m1();
mytruck.m2();
```

10. Suppose the `Truck` code from the previous problem changes to the following:

```
public class Truck extends Car {
    public void m1() {
        System.out.println("truck 1");
    }

    public void m2() {
        super.m1();
    }

    public String toString() {
        return super.toString() + super.toString();
    }
}
```

Using the same variables from the previous problem, what is the output from the following statements?

```
System.out.println(mytruck);
mytruck.m1();
mytruck.m2();
```

Section 9.3: Polymorphism

11. Using the A, B, C, and D classes from this section, what is the output of the following code fragment?

```
public static void main(String[] args) {
    A[] elements = {new B(), new D(), new A(), new C()};
    for (int i = 0; i < elements.length; i++) {
        elements[i].method2();
        System.out.println(elements[i]);
        elements[i].method1();
        System.out.println();
    }
}
```

12. Assume that the following classes have been defined:

```
1  public class Flute extends Blue {
2      public void method2() {
3          System.out.println("flute 2");
4      }
5
6      public String toString() {
7          return "flute";
8      }
9  }
```

```
1  public class Blue extends Moo {
2      public void method1() {
3          System.out.println("blue 1");
4      }
5  }
```

```
1  public class Shoe extends Flute {
2      public void method1() {
3          System.out.println("shoe 1");
4      }
5  }
```

```
1  public class Moo {
2      public void method1() {
3          System.out.println("moo 1");
4      }
```

```
5
6       public void method2() {
7           System.out.println("moo 2");
8       }
9
10      public String toString() {
11          return "moo";
12      }
13  }
```

What is the output produced by the following code fragment?

```
public static void main(String[] args) {
    Moo[] elements = {new Shoe(), new Flute(), new Moo(), new Blue()};
    for (int i = 0; i < elements.length; i++) {
        System.out.println(elements[i]);
        elements[i].method1();
        elements[i].method2();
        System.out.println();
    }
}
```

13. Using the classes from the previous problem, write the output that is produced by the following code fragment.

```
public static void main(String[] args) {
    Moo[] elements = {new Blue(), new Moo(), new Shoe(), new Flute()};
    for (int i = 0; i < elements.length; i++) {
        elements[i].method2();
        elements[i].method1();
        System.out.println(elements[i]);
        System.out.println();
    }
}
```

14. Assume that the following classes have been defined:

```
1  public class Mammal extends SeaCreature {
2      public void method1() {
3          System.out.println("warm-blooded");
4      }
5  }
```

```
1  public class SeaCreature {
2      public void method1() {
3          System.out.println("creature 1");
4      }
5
```

```
6       public void method2() {
7           System.out.println("creature 2");
8       }
9
10      public String toString() {
11          return "ocean-dwelling";
12      }
13  }
```

```
1   public class Whale extends Mammal {
2       public void method1() {
3           System.out.println("spout");
4       }
5
6       public String toString() {
7           return "BIG!";
8       }
9   }
```

```
1   public class Squid extends SeaCreature {
2       public void method2() {
3           System.out.println("tentacles");
4       }
5
6       public String toString() {
7           return "squid";
8       }
9   }
```

What output is produced by the following code fragment?

```
public static void main(String[] args) {
    SeaCreature[] elements = {new Squid(), new Whale(),
                              new SeaCreature(), new Mammal()};
    for (int i = 0; i < elements.length; i++) {
        System.out.println(elements[i]);
        elements[i].method1();
        elements[i].method2();
        System.out.println();
    }
}
```

15. Using the classes from the previous problem, write the output that is produced by the following code fragment:

```
public static void main(String[] args) {
    SeaCreature[] elements = {new SeaCreature(),
                              new Squid(), new Mammal(), new Whale()};
```

```
    for (int i = 0; i < elements.length; i++) {
        elements[i].method2();
        System.out.println(elements[i]);
        elements[i].method1();
        System.out.println();
    }
}
```

16. Assume that the following classes have been defined:

```
1  public class Bay extends Lake {
2      public void method1() {
3          System.out.print("Bay 1 ");
4          super.method2();
5      }
6      public void method2() {
7          System.out.print("Bay 2 ");
8      }
9  }
```

```
 1 public class Pond {
 2     public void method1() {
 3         System.out.print("Pond 1 ");
 4     }
 5     public void method2() {
 6         System.out.print("Pond 2 ");
 7     }
 8     public void method3() {
 9         System.out.print("Pond 3 ");
10     }
11 }
```

```
1  public class Ocean extends Bay {
2      public void method2() {
3          System.out.print("Ocean 2 ");
4      }
5  }
```

```
1  public class Lake extends Pond {
2      public void method3() {
3          System.out.print("Lake 3 ");
4          method2();
5      }
6  }
```

What output is produced by the following code fragment?

```
Pond[] ponds = {new Ocean(), new Pond(), new Lake(), new Bay()};
for (Pond p : ponds) {
```

```
        p.method1();
        System.out.println();
        p.method2();
        System.out.println();
        p.method3();
        System.out.println("\n");
    }
```

17. Suppose that the following variables referring to the classes from the previous problem are declared:

```
Pond var1 = new Bay();
Object var2 = new Ocean();
```

Which of the following statements produce compiler errors? For the statements that do not produce errors, what is the output of each statement?

```
((Lake) var1).method1();
((Bay) var1).method1();
((Pond) var2).method2();
((Lake) var2).method2();
((Ocean) var2).method3();
```

Section 9.4: Inheritance and Design

18. What is the difference between an is-a and a has-a relationship? How do you create a has-a relationship in your code?

19. Imagine a `Rectangle` class with objects that represent two-dimensional rectangles. The `Rectangle` has `width` and `height` fields with appropriate accessors and mutators, as well as `getArea` and `getPerimeter` methods.

You would like to add a `Square` class into your system. Is it a good design to make `Square` a subclass of `Rectangle`? Why or why not?

20. Imagine that you are going to write a program to play card games. Consider a design with a `Card` class and 52 subclasses, one for each of the unique playing cards (for example, `NineOfSpades` and `JackOfClubs`). Is this a good design? If so, why? If not, why not, and what might be a better design?

21. In Section 9.2 we discussed adding functionality for dividend payments to the `Stock` class. Why was it preferable to create a `DividendStock` class rather than editing the `Stock` class and adding this feature directly to it?

Section 9.5: Interfaces

22. What is the difference between implementing an interface and extending a class?

23. Consider the following interface and class:

```
public interface I {
    public void m1();
    public void m2();
}
public class C implements I {
    // code for class C
}
```

What must be true about the code for class `C` in order for that code to compile successfully?

24. What's wrong with the code for the following interface? What should be changed to make a valid interface for objects that have colors?

```
public interface Colored {
    private Color color;
    public Color getColor() {
        return color;
    }
}
```

25. Modify the `Point` class from Chapter 8 so that it implements the `Colored` interface and `Points` have colors. (You may wish to create a `ColoredPoint` class that extends `Point`.)

26. Declare a method called `getSideCount` in the `Shape` interface that returns the number of sides that the shape has. Implement the method in all shape classes. A circle is defined to have 0 sides.

Section 9.6: Case Study: Financial Class Hierarchy

27. What is an abstract class? How is an abstract class like a normal class, and how does it differ? How is it like an interface?

28. Consider the following abstract class and its subclass. What state and behavior do you know for sure will be present in the subclass? How do you know?

```
public abstract class Ordered {
    private String[] data;
    public void getElement(int i) {
        return data[i];
    }
    public abstract void arrange();
}

public class OrderedByLength extends Ordered {
    ...
}
```

29. Consider writing a program to be used to manage a collection of movies. There are three kinds of movies in the collection: dramas, comedies, and documentaries. The collector would like to keep track of each movie's title, the name of its director, and the year the movie was made. Some operations are to be implemented for all movies, and there will also be special operations for each of the three different kinds of movies. How would you design the class(es) to represent this system of movies?

Exercises

1. Write the class `Marketer` to accompany the other law firm classes described in this chapter. Marketers make $50,000 ($10,000 more than general employees) and have an additional method called `advertise` that prints `"Act now, while supplies last!"` Make sure to interact with the superclass as appropriate.

2. Write a class `Janitor` to accompany the other law firm classes described in this chapter. Janitors work twice as many hours per week as other employees (80 hours/week), they make $30,000 ($10,000 *less* than general employees), they get half as much vacation as other employees (only 5 days), and they have an additional method `clean` that prints `"Workin' for the man."` Make sure to interact with the superclass as appropriate.

3. Write a class `HarvardLawyer` to accompany the other law firm classes described in this chapter. Harvard lawyers are like normal lawyers, but they make 20% more money than a normal lawyer, they get 3 days more vacation, and they have to fill out four of the lawyer's forms to go on vacation. That is, the `getVacationForm` method should return `"pinkpinkpinkpink"`. Make sure to interact with the superclass as appropriate.

4. Write a class `MonsterTruck` that relates to the `Car` and `Truck` classes from Self-Check Problems 9 and 10 and whose methods have the following behavior. Whenever possible, use inheritance to reuse behavior from the superclasses.

Method	Output/Return
`m1`	`monster 1`
`m2`	`truck 1`
	`car 1`
`toString`	`"monster vroomvroom"`

5. For the next four problems, consider the task of representing types of tickets to campus events. Each ticket has a unique number and a price. There are three types of tickets: walk-up tickets, advance tickets, and student advance tickets. Figure 9.10 illustrates the types:

   - Walk-up tickets are purchased the day of the event and cost $50.
   - Advance tickets purchased 10 or more days before the event cost $30, and advance tickets purchased fewer than 10 days before the event cost $40.
   - Student advance tickets are sold at half the price of normal advance tickets: When they are purchased 10 or more days early they cost $15, and when they are purchased fewer than 10 days early they cost $20.

   Implement a class called `Ticket` that will serve as the superclass for all three types of tickets. Define all common operations in this class, and specify all differing operations in such a way that every subclass must implement them. No actual objects of type `Ticket` will be created. Each actual ticket will be an object of a subclass type. Define the following operations:

   - The ability to construct a ticket by number.
   - The ability to ask for a ticket's price.
   - The ability to `println` a ticket object as a `String`. An example `String` would be `"Number: 17, Price: 50.0"`.

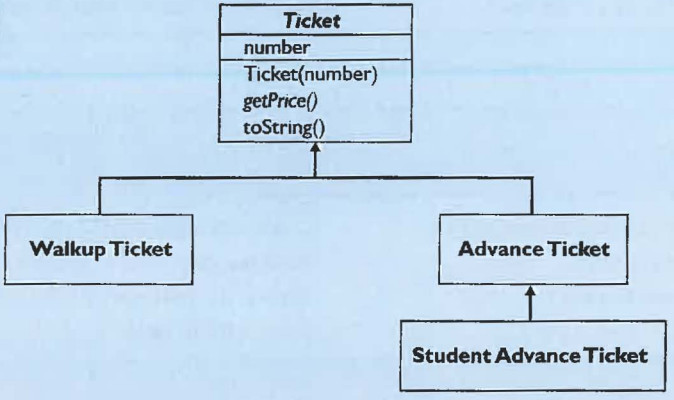


Figure 9.10 Classes of tickets that are available to campus events

6. Implement a class called `WalkupTicket` to represent a walk-up event ticket. Walk-up tickets are also constructed by number, and they have a price of $50.

7. Implement a class called `AdvanceTicket` to represent tickets purchased in advance. An advance ticket is constructed with a ticket number and with the number of days in advance that the ticket was purchased. Advance tickets purchased 10 or more days before the event cost $30, and advance tickets purchased fewer than 10 days before the event cost $40.

8. Implement a class called `StudentAdvanceTicket` to represent tickets purchased in advance by students. A student advance ticket is constructed with a ticket number and with the number of days in advance that the ticket was purchased. Student advance tickets purchased 10 or more days before the event cost $15, and student advance tickets purchased fewer than 10 days before the event cost $20 (half of a normal advance ticket). When a student advance ticket is printed, the `String` should mention that the student must show his or her student ID (for example, `"Number: 17, Price: 15.0 (ID required)"`).

9. `MinMaxAccount`. A company has written a large class `BankAccount` with many methods including:

`public BankAccount(Startup s)`	Constructs a `BankAccount` object using information in `s`
`public void debit(Debit d)`	Records the given debit
`public void credit(Credit c)`	Records the given credit
`public int getBalance()`	Returns the current balance in pennies

Design a new class `MinMaxAccount` whose instances can be used in place of a bank account but include new behavior of remembering the minimum and maximum balances ever recorded for the account. The class should have a constructor that accepts a `Startup` parameter. The bank account's constructor sets the initial balance on the basis of the startup information. Assume that only debits and credits change an account's balance. Include these new methods in your class:

`public int getMin()`	Returns the minimum balance in pennies
`public int getMax()`	Returns the maximum balance in pennies

10. `DiscountBill`. Suppose a class `GroceryBill` keeps track of a list of items being purchased at a market:

`public GroceryBill(Employee clerk)`	Constructs a grocery bill object for the given clerk
`public void add(Item i)`	Adds the given item to this bill
`public double getTotal()`	Returns the total cost of these items
`public void printReceipt()`	Prints a list of items

Grocery bills interact with `Item` objects, each of which has the public methods that follow. A candy bar item might cost 1.35 with a discount of 0.25 for preferred customers, meaning that preferred customers get it for 1.10. (Some

items will have no discount, 0.0.) Currently the preceding classes do not consider discounts. Every item in a bill is charged full price, and item discounts are ignored.

`public double getPrice()`	Returns the price for this item
`public double getDiscount()`	Returns the discount for this item

Define a class `DiscountBill` that extends `GroceryBill` to compute discounts for preferred customers. Its constructor accepts a parameter for whether the customer should get the discount. Your class should also adjust the total reported for preferred customers. For example, if the total would have been $80 but a preferred customer is getting $20 in discounts, then `getTotal` should report the total as $60 for that customer. Also keep track of the number of items on which a customer is getting a nonzero discount and the sum of these discounts, both as a total amount and as a percentage of the original bill. Include the extra methods that follow, which allow a client to ask about the discount. Return 0.0 if the customer is not a preferred customer or if no items were discounted.

`public DiscountBill(Employee clerk, boolean preferred)`	Constructs bill for given clerk
`public int getDiscountCount()`	Returns the number of items that were discounted, if any
`public double getDiscountAmount()`	Returns the total discount for this list of items, if any
`public double getDiscountPercent()`	Returns the percent of the total discount as a percent of what the total would have been otherwise

11. `FilteredAccount`. A cash processing company has a class called `Account` used to process transactions:

`public Account(Client c)`	Constructs an account using client information
`public boolean process(Transaction t)`	Processes the next transaction, returning `true` if the transaction was approved and `false` otherwise

Account objects interact with `Transaction` objects, which have many methods including

`public int value()`	Returns the value of this transaction in pennies (could be negative, positive or zero)

Design a new class called `FilteredAccount` whose instances can be used in place of normal accounts but which include the extra behavior of not processing transactions with a value of 0. More specifically, the new class should

indicate that a zero-valued transaction was approved but shouldn't call the `process` method for it. Your class should have a single constructor that accepts a parameter of type `Client`, and it should include the following method:

`public double percentFiltered()`	Returns the percent of transactions filtered out (between 0.0 and 100.0); returns 0.0 if no transactions are submitted

12. Add an `equals` method to the `TimeSpan` class introduced in Chapter 8. Two time spans are considered equal if they represent the same number of hours and minutes.
13. Add an `equals` method to the `Cash` class introduced in this chapter. Two cash objects are considered equal if they represent the same amount of money.
14. Add an `equals` method to each of the `Rectangle`, `Circle`, and `Triangle` classes introduced in this chapter. Two shapes are considered equal if their fields have equivalent values.
15. Write a class named `Octagon` whose objects represent regular octagons (eight-sided polygons). Your class should implement the `Shape` interface defined in this chapter, including methods for its area and perimeter. An `Octagon` object is defined by its side length. (You may need to search online to find formulas for the area and perimeter of a regular octagon.)
16. Write a class named `Hexagon` whose objects represent regular hexagons (6-sided polygons). Your class should implement the `Shape` interface defined in this chapter.
17. Declare an interface called `Incrementable` which represents items that store an integer that can be incremented in some way. The interface has a method called `increment` that increments the value and a method called `getValue` that returns the value. Once you have written the interface, write two classes called `SequentialIncrementer` and `RandomIncrementer` that implement the interface. The `SequentialIncrementer` begins its value at `0` and increases it by `1` each time it is incremented. The `RandomIncrementer` begins its value at a random integer and changes it to a new random integer each time it is incremented.

Programming Projects

1. Write an inheritance hierarchy of three-dimensional shapes. Make a top-level shape interface that has methods for getting information such as the volume and surface area of a three-dimensional shape. Then make classes and subclasses that implement various shapes such as cubes, rectangular prisms, spheres, triangular prisms, cones, and cylinders. Place common behavior in superclasses whenever possible, and use abstract classes as appropriate. Add methods to the subclasses to represent the unique behavior of each three-dimensional shape, such as a method to get a sphere's radius.
2. Write a set of classes that define the behavior of certain animals. They can be used in a simulation of a world with many animals moving around in it. Different kinds of animals will move in different ways (you are defining those differences). As the simulation runs, animals can "die" when two or more of them end up in the same location, in which case the simulator randomly selects one animal to survive the collision. See your course web site or www.buildingjavaprograms.com for supporting files to run such a simulation.

The following is an example set of animals and their respective behavior:

Class	`toString`	`getMove`
Bird	B	Moves randomly 1 step in one of the four directions each time
Frog	F	Moves randomly 3 steps in one of the four directions
Mouse	M	Moves west 1 step, north 1 step (zig-zag to the northwest)
Rabbit	V	Move north 2 steps, east 2 steps, south 2 steps ("hops" to the right)
Snake	S	Moves south 1 step, east 1 step, south 1 step, west 2 steps, south 1 step, east 3 steps, south 1 step, west 4 steps, ... ("slithers" left and right in increasing length)
Turtle	T	Moves south 5 steps, west 5 steps, north 5 steps, east 5 steps (clockwise box)
Wolf	W	Has custom behavior that you define

Your classes should be stored in files called Bird.java, Frog.java, Mouse.java, Rabbit.java, Snake.java, Turtle.java, and Wolf.java.

3. Write an inheritance hierarchy that stores data about sports players. Create a common superclass and/or interface to store information common to any player regardless of sport, such as name, number, and salary. Then create subclasses for players of your favorite sports, such as basketball, soccer, or tennis. Place sport-specific information and behavior (such as kicking or vertical jump height) into subclasses whenever possible.

4. Write an inheritance hierarchy to model items at a library. Include books, magazines, journal articles, videos, and electronic media such as CDs. Include in a superclass and/or interface common information that the library must have for every item, such as a unique identification number and title. Place behavior and information that is specific to items, such as a video's runtime length or a CD's musical genre, into the subclasses.