

```
1 // A class to represent employees in general.
2 public class Employee {
3     public void applyForVacation() {
4         System.out.println("Use the yellow vacation form.");
5     }
6
7     public void showHours() {
8         System.out.println("I work 40 hours per week.");
9     }
10
11     public void showSalary() {
12         System.out.println("My salary is $40,000.");
13     }
14
15     public void showVacation() {
16         System.out.println("I receive 2 weeks vacation.");
17     }
18 }
19
20 public class Lawyer extends Employee {
21     public void applyForVacation() {
22         System.out.println("Use the pink vacation form.");
23     }
24
25     public void showVacation() {
26         System.out.println("I receive 3 weeks vacation.");
27     }
28
29     public void sue() {
30         System.out.println("I'll see you in court!");
31     }
32 }
33
```

```

34 // A class to represent legal secretaries.
35 public class LegalSecretary extends Employee {
36     public void fileLegalBriefs() {
37         System.out.println("I could file all day!");
38     }
39
40     public void showSalary() {
41         System.out.println("My salary is $45,000.");
42     }
43 }
44
45 // A class to represent marketers.
46 public class Marketer extends Employee {
47     public void advertise() {
48         System.out.println("Act now, while supplies last!");
49     }
50
51     public void showSalary() {
52         System.out.println("My salary is $50,000.");
53     }
54 }
55
56 // A class to represent secretaries.
57 public class Secretary extends Employee {
58     public void takeDictation() {
59         System.out.println("I know how to take dictation.");
60     }
61 }

```

```

1 public class EmployeeMain {
2     public static void main(String[] args) {
3         System.out.println("Employee:");
4         Employee employee1 = new Employee();
5         employee1.applyForVacation();
6         employee1.showHours();
7         employee1.showSalary();
8         employee1.showVacation();
9         System.out.println();
10
11         System.out.println("Secretary:");
12         Secretary employee2 = new Secretary();
13         employee2.applyForVacation();
14         employee2.showHours();
15         employee2.showSalary();
16         employee2.showVacation();
17         employee2.takeDictation();
18     }
19 }
20
21 public class EmployeeMain3 {
22     public static void main(String[] args) {
23         Employee empl = new Employee();
24         Lawyer law = new Lawyer();
25         Marketer mark = new Marketer();
26         Secretary sec = new Secretary();
27
28         printInfo(empl);
29         printInfo(law);
30         printInfo(mark);
31         printInfo(sec);
32     }
33
34     public static void printInfo(Employee employee) { Polymorphism
35         employee.applyForVacation();
36         employee.showHours();
37         employee.showSalary();
38         employee.showVacation();
39         System.out.println();
40     }
41 }

```

Table 9.2 Sample instanceof Expressions


Expression	Result
<code>s instanceof String</code>	true
<code>s instanceof Point</code>	false
<code>p instanceof String</code>	false
<code>p instanceof Point</code>	true
<code>"hello" instanceof String</code>	true
<code>null instanceof Point</code>	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;
    }
}
```

 `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.

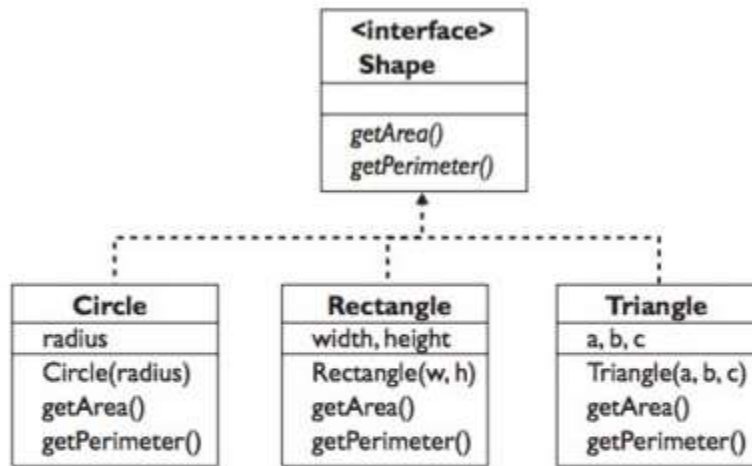
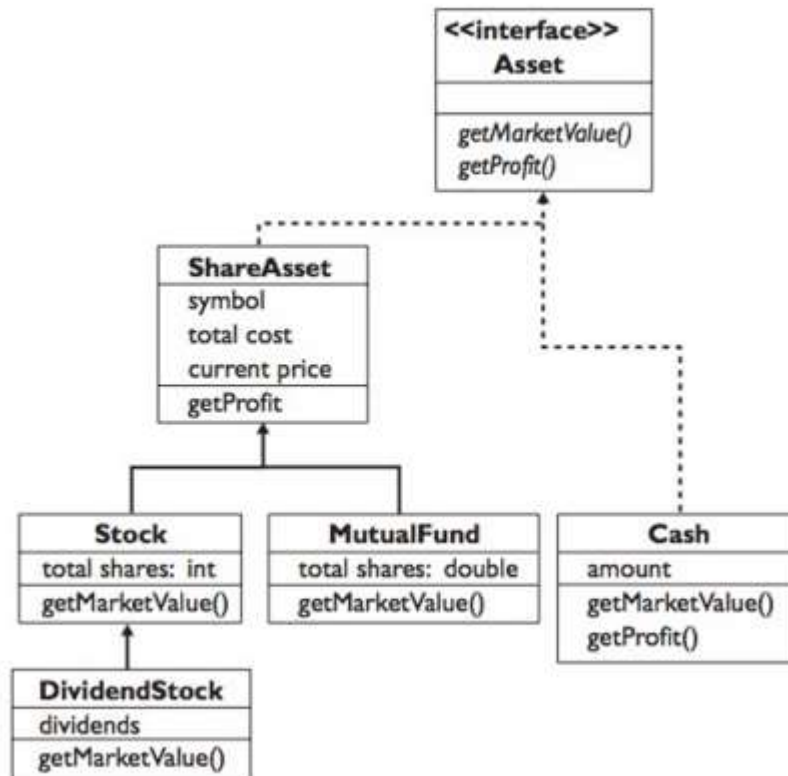


Figure 9.5 Hierarchy of shape classes

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 }
  
```



```

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    // stock at the given price per share.
25    public void purchase(int shares, double pricePerShare) {
26        totalShares += shares;
27        addCost(shares * pricePerShare);
28    }
29 }

```

```

30
31 // Represents a stock purchase that also pays dividends.
32 public class DividendStock extends Stock {
33     private double dividends;    // amount of dividends paid
34
35     // Constructs a new dividend stock with the given symbol
36     // and no shares purchased.
37     public DividendStock(String symbol, double currentPrice) {
38         super(symbol, currentPrice); // call Stock constructor
39         dividends = 0.0;
40     }
41
42     // Returns this DividendStock's market value, which is
43     // a normal stock's market value plus any dividends.
44     public double getMarketValue() {
45         return super.getMarketValue() + dividends;
46     }
47
48     // Records a dividend of the given amount per share.
49     public void payDividend(double amountPerShare) {
50         dividends += amountPerShare * getTotalShares();
51     }
52 }

```



```

54 // Represents a mutual fund asset.
55 public class MutualFund extends ShareAsset {
56     private double totalShares;
57
58     // Constructs a new MutualFund investment with the given
59     // symbol and price per share.
60     public MutualFund(String symbol, double currentPrice) {
61         super(symbol, currentPrice);
62         totalShares = 0.0;
63     }
64
65     // Returns the market value of this mutual fund, which
66     // is the number of shares times the price per share.
67     public double getMarketValue() {
68         return totalShares * getCurrentPrice();
69     }
70
71     // Returns the number of shares of this mutual fund.
72     public double getTotalShares() {
73         return totalShares;
74     }
75
76     // Records purchase of the given shares at the given price.
77     public void purchase(double shares, double pricePerShare) {
78         totalShares += shares;
79         addCost(shares * pricePerShare);
80     }
81 }

```



```
83 // Represents an amount of money held by an investor.
84 public class Cash implements Asset {
85     private double amount;    // amount of money held
86
87     // Constructs a cash investment of the given amount.
88     public Cash(double amount) {
89         this.amount = amount;
90     }
91
92     // Returns this cash investment's market value, which
93     // is equal to the amount of cash.
94     public double getMarketValue() {
95         return amount;
96     }
97
98     // Since cash is a fixed asset, it never has any profit.
99     public double getProfit() {
100         return 0.0;
101     }
102
103     // Sets the amount of cash invested to the given value.
104     public void setAmount(double amount) {
105         this.amount = amount;
106     }
107 }
```

```

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 }
9
10 // A general asset that has a symbol and holds shares.
11 public abstract class ShareAsset implements Asset {
12     private String symbol;
13     private double totalCost;
14     private double currentPrice;
15
16     // Constructs a new share asset with the given symbol
17     // and current price.
18     public ShareAsset(String symbol, double currentPrice) {
19         this.symbol = symbol;
20         this.currentPrice = currentPrice;
21         totalCost = 0.0;
22     }
23
24     // Adds a cost of the given amount to this asset.
25     public void addCost(double cost) {
26         totalCost += cost;
27     }
28
29     // Returns the price per share of this asset.
30     public double getCurrentPrice() {
31         return currentPrice;
32     }
33
34     // Returns the current market value of this asset.
35     public abstract double getMarketValue();
36
37     // Returns the profit earned on shares of this asset.
38     public double getProfit() {
39         // calls an abstract getMarketValue method
40         // (the subclass will provide its implementation)
41         return getMarketValue() - totalCost;
42     }
43
44     // Returns this asset's total cost spent on all shares.
45     public double getTotalCost() {
46         return totalCost;
47     }

```