CHAPTER 8

Polymorphism and Abstract Classes

8 Polymorphism and Abstract Classes

You know my methods, Watson.

Sir Arthur Conan Doyle, *The Crooked Man* (Sherlock Holmes)

Introduction

The three main programming mechanisms that constitute object-oriented programming (OOP) are encapsulation, inheritance, and polymorphism. We have already discussed the first two. In this chapter we discuss polymorphism. *Polymorphism* refers to the ability to associate many meanings to one method name by means of a special mechanism known as *late binding* or *dynamic binding.*

This chapter also covers *abstract classes*, which are classes in which some methods are not fully defined. Abstract classes are designed to be used only as base classes for defining new classes. You cannot create instances of (objects of) an abstract class; you can only create instances of its descendent classes.

Both polymorphism and abstract classes deal with code in which a method is used before it is defined. Although this may sound paradoxical, it all works out smoothly in Java.

Prerequisites

This chapter requires Chapters 1 through 5 and Chapter 7 with the exception that Section 5.4 on packages and `javadoc` is not required. This chapter does not use any material on arrays from Chapter 6.

Sections 8.1 on polymorphism and 8.2 on abstract classes are independent of each other and you may cover Section 8.2 before Section 8.1 if you wish.

8.1 Polymorphism

I did it my way.

Frank Sinatra

Inheritance allows you to define a base class and define software for the base class. That software can then be used not only for objects of the base class but also for objects of any class derived from the base class. *Polymorphism* allows you to make changes in the method definition for the derived classes and have those changes apply to the software written *for the base class.* This all happens automatically in Java but it is important to understand the process, and to

understand polymorphism we need a concrete example. The next subsection begins with such an example.

LATE BINDING

Suppose you are designing software for a graphics package that has classes for several kinds of figures, such as rectangles, circles, ovals, and so forth. Each figure might be an object of a different class. For example, the `Rectangle` class might have instance variables for a height, width, and center point, while the `Circle` class might have instance variables for a center point and a radius. In a well-designed programming project, all of these classes would be descendents of a single parent class called, for example, `Figure`. Now, suppose you want a method to draw a figure on the screen. To draw a circle, you need different instructions from those you need to draw a rectangle. So, each class needs to have a different method to draw its kind of figure. However, because the methods belong to the classes, they can all be called `draw`. If `r` is a `Rectangle` object and `c` is a `Circle` object, then `r.draw()` and `c.draw()` can be methods implemented with different code. All this is not new, but now we move on to something new.

Now, the parent class `Figure` may have methods that apply to all figures. For example, it might have a method called `center` that moves a figure to the center of the screen by erasing it and then redrawing it in the center of the screen. The method `center` of the class `Figure` might use the method `draw` to redraw the figure in the center of the screen. When you think of using the inherited method `center` with figures of the classes `Rectangle` and `Circle`, you begin to see that there are complications here.

To make the point clear and more dramatic, let's suppose the class `Figure` is already written and in use and at some later time you add a class for a brand-new kind of figure, say the class `Triangle`. Now `Triangle` can be a derived class of the class `Figure`, so the method `center` will be inherited from the class `Figure` and thus should apply to (and perform correctly for!) all `Triangle`s. But there is a complication. The method `center` uses `draw`, and the method `draw` is different for each type of figure. But, the method `center` is defined in the class `Figure` and so the method `center` was compiled before we wrote the code for the method `draw` of the class `Triangle`. When we invoke the method `center` with an object of the class `Triangle`, we want the code for the method `center` to use a method that was not even defined when we compiled the method `center`, namely the method `draw` for the class `Triangle`. Can this be made to happen in Java? Yes, it can, and moreover, it happens automatically. You need not do anything special when you define either the base class `Figure` or the derived class `Triangle`.

The situation we discussed for the method `center` in the derived class `Triangle` works out as we want because Java uses a mechanism known as **late binding** or **dynamic binding**. Let's see how late binding works in this case involving figure classes.

late binding

Binding refers to the process of associating a method definition with a method invocation. If the method definition is associated with the method definition when the code is compiled, that is called **early binding**. If the method definition is associated with the method invocation when the method is invoked (at run time), that is called **late binding** or **dynamic binding**. Java uses late binding for all methods except for a

binding

early binding

few cases discussed later in this chapter. Let's see how late binding works in the case of our method `center`.

Recall that the method `center` was defined in the class `Figure` and that the definition of the method `center` included an invocation of the method `draw`. If, contrary to fact, Java used early binding, then when the code for the method `center` was compiled the invocation of the method `draw` would be bound to the currently available definition of `draw`, which is the one given in the definition of `Figure`. If early binding were used, the method `center` would behave exactly the same for all derived classes of the class `Figure` as it does for objects created using the class `Figure`. But, fortunately, Java uses late binding, so when `center` is invoked by an object of the class `Triangle`, the invocation of `draw` (inside the method `center`) is not bound to a definition of `draw` until the invocation actually takes place. At that point in time, the run-time system knows the calling object is an instance of the class `Triangle` and so uses the definition of `draw` given in the definition of the class `Triangle` (even if the invocation of `draw` is inside the definition of the method `center`). So, the method `center` behaves differently for an object of the class `Triangle` than it would for an object that is just a plain old `Figure`. With late binding, as in Java, things automatically work out the way you normally want them to.

Note that in order for late binding to work, each object must somehow know which definition of each method applies to that object. So, when an object is created in a system using late binding, the description of the object must include (either directly or indirectly) a description of where the appropriate definition of each method is located. This additional overhead is the penalty you pay for the convenience of late binding.

Late Binding

With **late binding** the definition of a method is not bound to an invocation of the method until run time, in fact, not until the time at which the particular invocation takes place. Java uses late binding (for all methods except those discussed in the Pitfalls section entitled "No Late Binding for Static Methods ✤").

polymorphism

The terms **polymorphism** and *late binding* are essentially just different words for the same concept. The term *polymorphism* refers to the processes of assigning multiple meanings to the same method name using late binding.

Polymorphism

Polymorphism refers to the ability to associate many meanings to one method name by means of the late binding mechanism. Thus, polymorphism and late binding are really the same topic.

THE final MODIFIER

You can mark a method to indicate that it cannot be overridden with a new definition in a derived class. You do this by adding the `final` modifier to the method heading, as in the following sample heading:

final

```
public final void someMethod()
{
        .
        .
        .
```

An entire class can be declared final, in which case you cannot use it as a base class to derive any other class from it. The syntax for declaring a class to be final is illustrated in what follows:

```
public final class SomeClass
{
        .
        .
        .
```

If a method is marked as `final`, that means the compiler can use early binding with that particular method, which enables the compiler to be more efficient. However, the added efficiency is normally not great and we suggest not using the `final` modifier solely for reasons of efficiency. (Also, it can sometimes aid security to mark certain methods as `final`.)

You can view the `final` modifier as a way of turning off late binding for a method (or an entire class). Of course, it does more than just turn off late binding—it turns off the ability to redefine the method in any descendent class.

THE final MODIFIER

If you add the modifier `final` to the definition of a method, that indicates that the method may not be redefined in a derived class. If you add the modifier `final` to the definition of a class, that indicates that the class may not be used as a base class to derive other classes.

Example

SALES RECORDS

Suppose you are designing a record-keeping program for an automobile parts store. You want to make the program versatile, but you are not sure you can account for all possible situations. For example, you want to keep track of sales, but you cannot anticipate all types of sales. At first,

there will only be regular sales to retail customers who go to the store to buy one particular part. However, later you may want to add sales with discounts or mail order sales with a shipping charge. All of these sales will be for an item with a basic price and ultimately will produce some bill. For a simple sale, the bill is just the basic price, but if you later add discounts, then some kinds of bills will also depend on the size of the discount. Now your program needs to compute daily gross sales, which intuitively should just be the sum of all the individual sales bills. You may also want to calculate the largest and smallest sales of the day or the average sale for the day. All of these can be calculated from the individual bills, but many of the methods for computing the bills will not be added until later, when you decide what types of sales you will be dealing with. Because Java uses late binding, you can write a program to total all bills even though you will not determine the code for some of the bills until later. (For simplicity in this first example, we assume that each sale is for just one item, although we could, but will not here, account for sales of multiple items.)

Display 8.1 contains the definition for a class named `Sale`. All types of sales will be derived classes of the class `Sale`. The class `Sale` corresponds to simple sales of a single item with no added discounts and no added charges. Note that the methods `lessThan` and `equalDeals` both include invocations of the method `bill`. We can later define derived classes of the class `Sale` and define their versions of the method `bill`, and the definitions of the methods `lessThan` and `equalDeals`, which we gave with the class `Sale`, will use the version of the method `bill` that corresponds to the object of the derived class.

For example, Display 8.2 shows the derived class `DiscountSale`. Notice that the class `DiscountSale` requires a different definition for its version of the method `bill`. Now the methods `lessThan` and `equalDeals`, which use the method `bill`, are inherited from the base class `Sale`. But, when the methods `lessThan` and `equalDeals` are used with an object of the class `DiscountSale`, they will use the version of the method definition for `bill` that was given with the class `DiscountSale`. This is indeed a pretty fancy trick for Java to pull off. Consider the method call `d1.lessThan(d2)` for objects `d1` and `d2` of the class `DiscountSale`. The definition of the method `lessThan` (even for an object of the class `DiscountSale`) is given in the definition of the base class `Sale`, which was compiled before we ever even thought of the class `DiscountSale`. Yet, in the method call `d1.lessThan(d2)`, the line that calls the method `bill` knows enough to use the definition of the method `bill` given for the class `DiscountSale`. This all works out because Java uses late binding.

Display 8.3 gives a sample program that illustrates how the late binding of the method `bill` and the methods that use `bill` work in a complete program.

Self-Test Exercises

1. Explain the difference between the terms *late binding* and *polymorphism*.
2. Suppose you modify the definitions of the class `Sale` (Display 8.1) by adding the modifier `final` to the definition of the method `bill`. How would that change the output of the program in Display 8.3?

Display 8.1 The Base Class Sale *(Part 1 of 3)*

```
1  /**
2   Class for a simple sale of one item with no tax, discount, or other adjustments.
3   Class invariant: The price is always nonnegative; the name is a nonempty string.
4  */
5  public class Sale
6  {
7      private String name; //A nonempty string
8      private double price; //nonnegative

9      public Sale()
10     {
11         name = "No name yet";
12         price = 0;
13     }

14     /**
15      Precondition: theName is a nonempty string; thePrice is nonnegative.
16     */
17     public Sale(String theName, double thePrice)
18     {
19         setName(theName);
20         setPrice(thePrice);
21     }

22     public Sale(Sale originalObject)
23     {
24         if (originalObject == null)
25         {
26             System.out.println("Error: null Sale object.");
27             System.exit(0);
28         }
29         //else
30         name = originalObject.name;
31         price = originalObject.price;
32     }

33     public static void announcement()
34     {
35         System.out.println("This is the Sale class.");
36     }

37     public double getPrice()
38     {
39         return price;
40     }
```

Display 8.1 **The Base Class Sale** ***(Part 2 of 3)***

```
41      /**
42        Precondition: newPrice is nonnegative.
43       */
44       public void setPrice(double newPrice)
45       {
46           if (newPrice >= 0)
47               price = newPrice;
48           else
49           {
50               System.out.println("Error: Negative price.");
51               System.exit(0);
52           }
53       }

54       public String getName()
55       {
56           return name;
57       }

58       /**
59        Precondition: newName is a nonempty string.
60       */
61       public void setName(String newName)
62       {
63           if (newName != null && newName != "")
64               name = newName;
65           else
66           {
67               System.out.println("Error: Improper name value.");
68               System.exit(0);
69           }
70       }

71       public String toString()
72       {
73           return (name + " Price and total cost = $" + price);
74       }

75       public double bill()
76       {
77           return price;
78       }
```

Display 8.1 The Base Class Sale *(Part 3 of 3)*

```
79      /*
80       Returns true if the names are the same and the bill for the calling
81       object is equal to the bill for otherSale; otherwise returns false.
82       Also returns false if otherObject is null.
83      */
84      public boolean equalDeals(Sale otherSale)
85      {
86          if (otherSale == null)
87              return false;
88          else
89              return (name.equals(otherSale.name)
90                  && bill() == otherSale.bill());
91      }

92      /*
93       Returns true if the bill for the calling object is less
94       than the bill for otherSale; otherwise returns false.
95      */
96      public boolean lessThan (Sale otherSale)
97      {
98          if (otherSale == null)
99          {
100             System.out.println("Error: null Sale object.");
101             System.exit(0);
102         }
103         //else
104         return (bill() < otherSale.bill());
105     }

106     public boolean equals(Object otherObject)
107     {
108         if (otherObject == null)
109             return false;
110         else if (getClass() != otherObject.getClass())
111             return false;
112         else
113         {
114             Sale otherSale = (Sale)otherObject;
115             return (name.equals(otherSale.name)
116                && (price == otherSale.price));
117         }
118     }
119 }
```

When invoked, these methods will use the definition of the method `bill` that is appropriate for each of the objects.

Display 8.2 The Derived Class DiscountSale *(Part 1 of 2)*


```
1  /**
2   Class for a sale of one item with discount expressed as a percent of the price,
3   but no other adjustments.
4   Class invariant: The price is always nonnegative; the name is a
5   nonempty string; the discount is always nonnegative.
6  */
```

```
1  public class DiscountSale extends Sale
2  {
3      private double discount; //A percent of the price. Cannot be negative.

4      public DiscountSale()
5      {
6          super();
7          discount = 0;
8      }

9      /**
10      Precondition: theName is a nonempty string; thePrice is nonnegative;
11      theDiscount is expressed as a percent of the price and is nonnegative.
12     */
13     public DiscountSale(String theName,
14                                 double thePrice, double theDiscount)
15     {
16         super(theName, thePrice);
17         setDiscount(theDiscount);
18     }

19     public DiscountSale(DiscountSale originalObject)
20     {
21         super(originalObject);
22         discount = originalObject.discount;
23     }

24     public static void announcement()
25     {
26         System.out.println("This is the DiscountSale class.");
27     }

28     public double bill()
29     {
30         double fraction = discount/100;
31         return (1 - fraction)*getPrice();
32     }
```

The meaning would be unchanged if this line were omitted.

Display 8.2 The Derived Class `DiscountSale` *(Part 2 of 2)*

```
33      public double getDiscount()
34      {
35          return discount;
36      }

37      /**
38       Precondition: Discount is nonnegative.
39      */
40      public void setDiscount(double newDiscount)
41      {
42          if (newDiscount >= 0)
43              discount = newDiscount;
44          else
45          {
46              System.out.println("Error: Negative discount.");
47              System.exit(0);
48          }
49      }

50      public String toString()
51      {
52          return (getName() + " Price = $" + getPrice()
53                  + " Discount = " + discount + "%\n"
54                  + "   Total cost = $" + bill());
55      }

56      public boolean equals(Object otherObject)
      <The rest of the definition of equals is Self-Test Exercise 4.>

57  }
```

3. Would it be legal to add the following method definition to the class `DiscountSale`?

```
public static boolean isAGoodBuy(Sale theSale)
{
    return (theSale.getDiscount() > 20);
}
```

4. Complete the definition of the method `equals` for the class `DiscountSale` (Display 8.2).

Display 8.3 Late Binding Demonstration


```
1  /**
2   Demonstrates late binding.
3  */
4  public class LateBindingDemo
5  {
6      public static void main(String[] args)
7      {
8          Sale simple = new Sale("floor mat", 10.00);//One item at $10.00.
9          DiscountSale discount = new DiscountSale("floor mat", 11.00, 10);
10                                  //One item at $11.00 with a 10% discount.
11         System.out.println(simple);
12         System.out.println(discount);

13         if (discount.lessThan(simple))
14             System.out.println("Discounted item is cheaper.");
15         else
16             System.out.println("Discounted item is not cheaper.");

17         Sale regularPrice = new Sale("cup holder", 9.90);//One item at $9.90.
18         DiscountSale specialPrice = new DiscountSale("cup holder", 11.00, 10);
19                                  //One item at $11.00 with a 10% discount.
20         System.out.println(regularPrice);
21         System.out.println(specialPrice);

22         if (specialPrice.equalDeals(regularPrice))
23             System.out.println("Deals are equal.");
24         else
25             System.out.println("Deals are not equal.");
26     }
27 }
```

The method `lessThan` uses different definitions for `discount.bill()` and `simple.bill()`.

The method `equalDeals` uses different definitions for `specialPrice.bill()` and `regularPrice.bill()`.

The `equalDeals` method says that two items are equal provided they have the same name and the same bill (same total cost). It does not matter how the bill (the total cost) is calculated.

Sample Dialogue

```
floor mat Price and total cost = $10.0
floor mat Price = $11.0 Discount = 10.0%
   Total cost = $9.9
Discounted item is cheaper.
cup holder Price and total cost = $9.9
cup holder Price = $11.0 Discount = 10.0%
   Total cost = $9.9
Deals are equal.
```

LATE BINDING WITH `toString`

In the subsection "The Methods `equals` and `toString`" in Chapter 4, we noted that if you include an appropriate `toString` method in the definition of a class, then you can output an object of the class using `System.out.println`. For example, the following works because `Sale` has a suitable `toString` method:

```
Sale aSale = new Sale("tire gauge", 9.95);
System.out.println(aSale);
```

This produces the screen output

```
tire gauge Price and total cost = $9.95
```

This happens because Java uses late binding. Here are the details.

The method invocation `System.out.println(aSale)` is an invocation of the method `println` with the calling object `System.out`. One definition of the method `println` has a single argument of type `Object`. The definition is equivalent to the following:

```
public void println(Object theObject)
{
    System.out.println(theObject.toString());
}
```

(The invocation of the method `println` inside the braces is a different, overloaded definition of the method `println`. That invocation inside the braces uses a method `println` that has a parameter of type `String`, not a parameter of type `Object`.)

This definition of `println` was given before the class `Sale` was defined. Yet in the invocation

```
System.out.println(aSale);
```

with an argument `aSale` of type `Sale` (and hence also of type `Object`), it is the definition of `toString` in the class `Sale` that is used, not the definition of `toString` in the class `Object`. Late binding is what makes this work.

Pitfall

NO LATE BINDING FOR STATIC METHODS

Java does not use late binding with private methods, methods marked `final`, or static methods. With private methods and `final` methods, this is not an issue since dynamic binding would serve no purpose anyway. However, with static methods it can make a difference when the static method is invoked using a calling object, and such cases arise more often than you might think.

When Java (or any language) does not use late binding, it uses **static binding**. With static binding, the decision of which definition of a method to use with a calling object is made at compile time based on the type of the variable naming the object.

Display 8.4 illustrates the effect of static binding on a static method with a calling object. Note that the static method `announcement()` in the class `Sale` has its definition overridden in the derived class `DiscountSale`. However, when an object of type `DiscountSale` is named by a variable of type `Sale`, it is the definition `announcement()` in the class `Sale` that is used, not the definition of `announcement` in the class `DiscountSale`.

"So, what's the big deal?" you may ask. A static method is normally called with a class name and not a calling object. It may look that way, but there are cases where a static method has a calling object in an inconspicuous way. If you invoke a static method within the definition of a nonstatic method but without any class name or calling object, then the calling object is an implicit `this`, which is a calling object.

For example, suppose you add the following method to the class `Sale`:

```
public void showAdvertisement()
{
    announcement();
    System.out.println(toString( ));
}
```

Suppose further that the method `showAdvertisement` is not overridden in the class `DiscountSale`, then the method `showAdvertisement` is inherited unchanged from `Sale`.

Now consider the following code:

```
Sale s = new Sale("floor mat", 10.00);
DiscountSale discount = new DiscountSale("floor mat", 11.00, 10);
s.showAdertisement();
discount.showAdertisement();
```

You might expect the following output:

```
This is the Sale class.
floor mat Price and total cost = $10.0
This is the DiscountSale class.
floor mat Price = $11.0 Discount = 10.0%
   Total cost = $9.9
```

However, since the definition used for the static method `announcement`, inside of `showAdvertisement`, is determined at compile time (based on the type of the variable holding the calling object), the output actually is the following, where the change is shown in red:

```
This is the Sale class.
floor mat Price and total cost = $10.0
This is the Sale class.
floor mat Price = $11.0 Discount = 10.0%
   Total cost = $9.9
```

Java uses late binding with the nonstatic method `toString` but static binding with the static method `announcement`.

Display 8.4 No Late Binding with Static Methods

```
1  /**
2   Demonstrates that static methods use static binding.
3  */
4  public class StaticMethodsDemo
5  {
6      public static void main(String[] args)
7      {
8          Sale.announcement();
9          DiscountSale.announcement();
10         System.out.println(
11              "That showed that you can override a static method definition.");

12         Sale s = new Sale();
13         DiscountSale discount = new DiscountSale();
14         s.announcement();
15         discount.announcement();
16         System.out.println("No surprises so far, but wait.");

17         Sale discount2 = discount;
18         System.out.println(
19               "discount2 is a DiscountSale object in a Sale variable.");
20         System.out.println( "Which definition of announcement() will it use?");
21         discount2.announcement();
22         System.out.println(
23               "It used the Sale version of announcement()!");
24     }
25 }
```

Java uses static binding with static methods so the choice of which definition of a static method to use is determined by the type of the variable, not by the object.

`discount` and `discount2` name the same object, but one is a variable of type `Sale` and one is a variable of type `DiscountSale`.

Sample Dialogue

```
This is the Sale class.
This is the DiscountSale class.
That showed that you can override a static method definition.
This is the Sale class.
This is the DiscountSale class.
No surprises so far, but wait.
discount2 is a DiscountSale object in a Sale variable.
Which definition of announcement() will it use?
This is the Sale class.
It used the Sale version of announcement()!
```

If Java had used late binding with static methods, then this would have been the other announcement.

DOWNCASTING AND UPCASTING

The following is perfectly legal (given the class definitions in Displays 8.1 and 8.2):

```
Sale saleVariable;
DiscountSale discountVariable =
                  new DiscountSale("paint", 15, 10);
saleVariable = discountVariable;
System.out.println(saleVariable.toString());
```

An object of a derived class, in this case the derived class `DiscountSale`, also has the type of its base class, in this case `Sale`, and so can be assigned to a variable of the base class type. Now let's consider the invocation of the method `toString()` on the last line of this code.

Because Java uses late binding, the invocation

```
saleVariable.toString()
```

uses the definition of the method `toString` given in the class `DiscountSale`. So the output is

```
paint Price = $15.0 Discount = 10.0%
   Total cost = $13.5
```

Because of late binding, the meaning of the method `toString` is determined by the object, not by the type of the variable `saleVariable`.

You may well respond, "Who cares? Why would I ever want to assign an object of type `DiscountSale` to a variable of type `Sale`?"[1] You make such assignments more often than you might think, but you tend to not notice them because they happen behind the scenes. Recall that a parameter is really a local variable, so every time you use an argument of type `DiscountSale` for a parameter of type `Sale`, you are assigning an object of type `DiscountSale` to a variable of type `Sale`. For example, consider the following invocation taken from the definition of the copy constructor for `DiscountSale` (Display 8.2):

```
super(originalObject);
```

In this invocation `originalObject` is of type `DiscountSale`, but `super` is the copy constructor for the base class `Sale` and so `super` has a parameter of type `Sale`, which is a local variable of type `Sale` that is set equal to the argument `originalObject` of type `DiscountSale`.

Note that the type of the variable naming an object determines which method names can be used in an invocation with that calling object. (Self-Test Exercise 3 may

[1] It is actually the references to the object that are assigned, not the objects themselves, but that subtlety is not relevant to what we are discussing here and the language is already complicated enough.

help you to understand this point.) However, the object itself always determines the meaning of a method invocation performed by an object; this is simply what we mean by late binding.

An Object Knows the Definitions of Its Methods

The type of a class variable determines which method names can be used with the variable, but the object named by the variable determines which definition of the method name is used. A special case of this rule is the following: The type of a class parameter determines which method names can be used with the parameter, but the argument determines which definition of the method name is used.

Assigning an object of a derived class to a variable of a base class (or any ancestor class) is often called **upcasting** because it is like a type cast to the type of the base class and, in the normal way of writing inheritance diagrams base classes are drawn above derived classes.[2]

upcasting

When you do a type cast from a base case to a derived class (or from any ancestor class to any descendent class), that is called a **downcast**. Upcasting is pretty straightforward; there are no funny cases to worry about, and in Java things always work out the way you want them to. Downcasting is more troublesome. First of all, downcasting does not always make sense. For example, the downcast

downcasting

```
Sale saleVariable = new Sale("paint", 15);
DiscountSale discountVariable;
discountVariable = (DiscountSale)saleVariable;//Error
```

does not make sense because the object named by `saleVariable` has no instance variable named `discount` and so cannot be an object of type `DiscountSale`. Every `DiscountSale` is a `Sale`, but not every `Sale` is a `DiscountSale`, as indicated by this example. It is your responsibility to use downcasting only in situations where it makes sense.

It is instructive to note that

```
discountVariable = (DiscountSale)saleVariable;
```

produces a run-time error but will compile with no error. However, the following, which is also illegal, produces a compile-time error:

```
discountVariable = saleVariable;
```

---

[2] We prefer to think of an object of the derived class as actually having the type of its base class as well as its own type. So, this is not, strictly speaking, a type cast but it does no harm to follow standard usage and call it a type cast in this case.

Java catches these downcasting errors as soon as it can, which may be at compile time or at run time depending on the case.

While downcasting can be dangerous, it is sometimes necessary. For example, we inevitably use downcasting when we define an `equals` method for a class. For example, note the following line from the definition of `equals` in the class `Sale` (Display 8.1):

```
Sale otherSale = (Sale)otherObject;
```

This is a downcast from the type `Object` to the type `Sale`. Without this downcast, the instance variables `name` and `price` in the `return` statement, reproduced below, would be illegal, since the class `Object` has no such instance variables:

```
return (name.equals(otherSale.name)
         && (price == otherSale.price));
```

Pitfall

Downcasting

It is the responsibility of you the programmer to use downcasting only in situations where it makes sense. The compiler makes no checks to see if downcasting is reasonable. However, if you use downcasting in a situation in which it does not make sense, you will usually get a run-time error message.

Tip

Checking to See If Downcasting Is Legitimate

You can use the `instanceof` operator to test whether or not a downcasting is sensible. A downcasting to a specific type is sensible if the object being cast is an instance of that type, and that is exactly what the `instanceof` operator tests for.

instanceof

The `instanceof` operator checks if an object is of the type given as its second argument. The syntax is

```
Object instanceof Class_Name
```

which returns `true` if *Object* is of type *Class_Name*; otherwise it returns `false`. So, the following will return `true` if `someObject` is of type `DiscountSale`:

```
someObject instanceof DiscountSale
```

Note that since every object of every descendent class of `DiscountSale` is also of type `DiscountSale`, this expression will return `true` if `someObject` is an instance of any descendent class of `DiscountSale`.

So, if you want to type cast to DiscountSale, then you can make the casts safer as follows:

```
DiscountSale ds = new DiscountSale();
if (someObject instanceof DiscountSale)
{
    ds = (DiscountSale)someObject;
    System.out.println("ds was changed to " + someObject);
}
else
    System.out.println("ds was not changed.");
```

someObject might be, for example, a variable of type Sale or of type Object.

Self-Test Exercises

5. Consider the following code, which is identical to the code discussed in the opening of the previous subsection except that we have added the type cast shown in color:

```
Sale saleVariable;
DiscountSale discountVariable =
                new DiscountSale("paint", 15, 10);
saleVariable = (Sale)discountVariable;
System.out.println(saleVariable.toString());
```

   We saw that without the type cast the definition of the toString method used is the one given in the definition of the class DiscountSale. With this added type cast, will the definition of the toString method used still be the one given in DiscountSale or will it be the one given in the definition of Sale?

6. Would it be legal to add the following method definition to the class DiscountSale? What about adding it to the class Sale?

```
public static void showDiscount(Sale object)
{
    System.out.println("Discount = "
                      + object.getDiscount());
}
```

7. ✣ What output is produced by the following code?

```
Sale someObject = new DiscountSale("map", 5, 0);
DiscountSale ds = new DiscountSale();
if (someObject instanceof DiscountSale)
{
    ds = (DiscountSale)someObject;
    System.out.println("ds was changed to " + someObject);
```

```
}
else
    System.out.println("ds was not changed.");
```

8. ✣ What output is produced by the following code?

```
Sale someObject = new Sale("map", 5);
DiscountSale ds = new DiscountSale();
if (someObject instanceof DiscountSale)
{
    ds = (DiscountSale)someObject;
    System.out.println("ds was changed to " + someObject);
}
else
    System.out.println("ds was not changed.");
```

9. ✣ Suppose we removed the qualifier `static` from the method `announcement()` in both `Sale` (Display 8.1) and `DiscountSale` (Display 8.2). What would be the output produced by the following code (which is similar to the end of Display 8.4)?

```
Sale s = new Sale( );
DiscountSale discount = new DiscountSale();
s.announcement( );
discount.announcement( );
System.out.println("No surprises so far, but wait.");

Sale discount2 = discount;
System.out.println(
   "discount2 is a DiscountSale object in a Sale variable.");
System.out.println(
     "Which definition of announcement( ) will it use?");
discount2.announcement( );
System.out.println(
       "Did it used the Sale version of announcement( )?");
```

A FIRST LOOK AT THE `clone` METHOD

Every object inherits a method named `clone` from the class `Object`. The method `clone` has no parameters and is supposed to return a copy of the calling object. However, the inherited version of `clone` was not designed to be used as is. Instead, you are expected to override the definition of `clone` with a version appropriate for the class you are defining. In Chapter 13 we will describe the officially sanctioned way to define the method `clone`. The officially sanctioned way turns out to be a bit complicated and requires material we do not cover until Chapter 13. In this section we will describe a simple way to define `clone` that will work in most situations and allow us to discuss how polymorphism interacts with the `clone` method. If you are in a hurry to see the

officially correct way to define clone, you can read Chapter 13 immediately after this section (Section 8.1) with no loss of continuity in your reading.

The method clone has no parameters and should return a copy of the calling object. The returned object should have identical data to that of the calling object, but it normally should be a different object (an identical twin or "a clone"). You usually want the clone method to return the same kind of copy as what we have been defining for copy constructors, which is what is known as a *deep copy*. (You many want to review the subsection entitled "Copy Constructors" in Chapter 5.)

A clone method serves very much the same purpose as a copy constructor but, as you will see in an upcoming subsection, there are situations where a clone method works as you want whereas a copy constructor does not perform as desired.

As with other methods inherited from the class Object, the method clone needs to be redefined (overridden) before it performs properly.

The heading for the method clone in the class Object, and hence the correct heading for the method clone in any class you define, is as follows[3]:

```
public Object clone()
```

You can use the copy constructor to complete the definition of the clone method as follows for the class Sale in Display 8.1:

```
public Object clone()
{
    return new Sale(this );
}
```

This is not the officially sanctioned way to define a clone method, and in fact the Java documentation says you should not define it this way. However, it does work correctly and some authorities say it is acceptable. In Chapter 13 we will discuss the officially sanctioned way of defining the method clone when we introduce the Cloneable interface.

Note that although the method clone for the class Sale returns a copy of an object of the class Sale, it always returns it as an object of type Object. For example, consider the class Sale in Display 8.1. If we add our definition of the clone method to the class Sale, you can make a copy of an object of type Sale as follows:

```
Sale original = new Sale("tire gauge", 9.95);
Sale copy = (Sale)original.clone();
```

Be sure to notice the type cast (Sale).

---

[3] In the class Object, the method clone is protected, not public, but it normally makes more sense to make it public in the classes you define.

The `clone` method for the `DiscountSale` class can be defined similarly:

```
public Object clone()
{
    return new DiscountSale(this );
}
```

extra code on CD

The definitions of the classes `Sale` and `DiscountSale` on the CD that accompanies this book each include the method `clone` defined as we just decribed.

Pitfall

The `clone` Method Return Type Is `Object`

You might be tempted to write the `clone` method for the class `Sale` (discussed in the previous subsection) with a return type of `Sale`, like so:

```
public Sale clone()
{
    return new Sale(this);
}
```

If you do this, it will produce a compiler error message. The reason for the error message is that there is an inherited method with the heading

```
public Object clone()
```

So, this definition would be overriding this inherited method, and when you override a method definition, you cannot change the return type of the method.

Pitfall

Limitations of Copy Constructors

Copy constructors work well in most simple cases. However, there are situations where they do not, indeed cannot, do their job. That is why Java favors using the method `clone` in place of using a copy constructor. Here's a simple example of where the copy constructor does not do its job, but the `clone` method does.

For this discussion assume that the classes `Sale` and `DiscountSale` each have a `clone` method added. The definitions of these `clone` methods are given in the previous subsection.

Suppose you have a method with the following heading (the methods Sale and DiscountSale were defined in Displays 8.1 and 8.2):

```
/**
 Supposedly returns a safe copy of a. That is, if b is the
 array returned, then b[i] is supposedly an independent copy of a[1].
*/
public static Sale[] badCopy(Sale[] a)
{
    Sale[] b = new Sale[a.length];
    for (int i = 0; i < a.length; i++)
        b[i] = new Sale(a[i]);//Problem here!
    return b;
}
```

Now if your array a contains objects from derived classes of Sale, such as objects of type DiscountSale, then badCopy(a) will not return a true copy of a. Every element of the array badCopy(a) will be a plain old Sale, since the Sale copy constructor only produces plain old Sale objects; no element in badCopy(a) will be an instance of the class DiscountSale.

If we instead use the method clone, things work out as they should; the following is the correct way to define our copy method:

```
public static Sale[] goodCopy(Sale[] a)
{
    Sale[] b = new Sale[a.length];
    for (int i = 0; i < a.length; i++)
        b[i] = (Sale)(a[i].clone( ));
    return b;
}
```

Because of late binding (polymorphism), a[i].clone() always means the correct version of the clone method. If a[i] is an object created with a constructor of the class DiscountSale, a[i].clone() will invoke the definition of clone() given in the definition of the class DiscountSale. If a[i] is an object created with a constructor of the class Sale, a[i].clone() will invoke the definition of clone() given in the definition of the class Sale. (The reason for the type cast to Sale is that clone returns its value as type Object. This type cast has nothing to do with the issue under discussion.) This is illustrated in Display 8.5.

This may seem like a sleight of hand. After all, in the classes Sale and DiscountSale we defined the method clone in terms of copy constructors. We reproduce the definitions of clone from the class Sale and DiscountSale below:

```
//For Sale class
public Object clone()
{
```

Display 8.5 Copy Constructor Versus `clone` Method *(Part 1 of 2)*

This program assumes that a `clone` method has been added to the class `Sale` and to the class `DiscountSale`.

```
1  /**
2   Demonstrates where the clone method works,
3   but copy constructors do not.
4  */
5  public class CopyingDemo
6  {

7      public static void main(String[] args)
8      {
9          Sale[] a = new Sale[2];
10         a[0] = new Sale("atomic coffee mug", 130.00);
11         a[1] = new DiscountSale("invisible paint", 5.00, 10);
12         int i;

13         Sale[] b = badCopy(a);

14         System.out.println("With copy constructors:");
15         for (i = 0; i < a.length; i++)
16         {
17             System.out.println("a[" + i + "] = " + a[i]);
18             System.out.println("b[" + i + "] = " + b[i]);
19             System.out.println();
20         }
21         System.out.println();

22         b = goodCopy(a);

23         System.out.println("With clone method:");
24         for (i = 0; i < a.length; i++)
25         {
26             System.out.println("a[" + i + "] = " + a[i]);
27             System.out.println("b[" + i + "] = " + b[i]);
28             System.out.println();
29         }

30     }

31     /**
32      Supposedly returns a safe copy of a. That is, if b is the
33      array returned, then b[i] is supposedly an independent copy of a[1].
34     */
```

Display 8.5 **Copy Constructor Versus `clone` Method** ***(Part 2 of 2)***

```
35      public static Sale[] badCopy(Sale[] a)
36      {
37          Sale[] b = new Sale[a.length];
38          for (int i = 0; i < a.length; i++)
39                b[i] = new Sale(a[i]);//Problem here!
40          return b;
41      }
42
43      /**
44       Returns a safe copy of a.That is, if b is the
45       array returned, then b[i] is an independent copy of a[1].
46      */
47      public static Sale[] goodCopy(Sale[] a)
48      {
49          Sale[] b = new Sale[a.length];
50          for (int i = 0; i < a.length; i++)
51                b[i] = (Sale)(a[i].clone( ));
52          return b;
53      }
54  }
```

Sample Dialogue

```
With copy constructors:
a[0] = atomic coffee mug Price and total cost = $130.0
b[0] = atomic coffee mug Price and total cost = $130.0

a[1] = invisible paint Price = $5.0 Discount 10.0%
   Total cost = $4.5
b[1] = invisible paint Price and total cost = $5.0

With clone method:
a[0] = atomic coffee mug Price and total cost = $130.0
b[0] = atomic coffee mug Price and total cost = $130.0

a[1] = invisible paint Price = $5.0 Discount 10.0%
   Total cost = $4.5
b[1] = invisible paint Price = $5.0 Discount 10.0%
   Total cost = $4.5
```

The copy constructor lost the discount.

The `clone` method did not lose the discount.

```
        return new Sale(this);
    }

    //For DiscountSale class
    public Object clone()
    {
        return new DiscountSale(this);
    }
```

So, why is using the method `clone` any different than using a copy constructor? The difference is simply that the method creating the copy of an element `a[i]` has the same name `clone` in all the classes, and polymorphism works with method names. The copy constructors named `Sale` and `DiscountSale` have different names, and polymorphism has nothing to do with methods of different names.

We will have more to say about the `clone` method in Chapter 13 when we discuss the `Cloneable` interface.

8.2 Abstract Classes

> *It is for us, the living, rather to be dedicated here to the unfinished work which they who fought here have thus far so nobly advanced.*
>
> Abraham Lincoln, *Gettysburg Address*

An abstract class is a class that has some methods without complete definitions. You cannot create an object using an abstract class constructor, but you can use an abstract class as a base class to define a derived class.

ABSTRACT CLASSES

In Chapter 7 we defined a class named `Employee` and two of its derived classes, `HourlyEmployee` and `SalariedEmployee`. Display 8.6 repeats the details of these class definitions, which we will use in this discussion.

Suppose that when we defined the class `Employee` we knew that we were going to frequently compare employees to see if they have the same pay. We might have added the following method to the class `Employee`:

```
public boolean samePay(Employee other)
{
    return (this.getPay() == other.getPay());
}
```

Display 8.6 Employee Class and Its Derived Classes *(Part 1 of 2)*

These show the details needed for the current discussion. You should not need to review the entire class definitions from Chapter 7. Complete definitions of all these classes are given in the subdirectory for this chapter on the CD that comes with this text.

extra code on CD

```
1  public class Employee
2  {
3      private String name;
4      private Date hireDate;

5      public Employee()
```

The class `Date` is defined in Display 4.11, but the details are not important to the current discussion. There is no need to review the definition of the class `Date`.

<The body of the constructor is given in Display 7.2,
but the details are not needed for this discussion.>

```
6      public boolean equals(Object otherObject)
```

<The body of the method `equals` is the same as in Display 7.8 of Chapter 7, but the details of the definition are not important to the current discussion.>

<All other constructor and other method definitions are exactly the same as in Display 7.2.>

The class `Employee` has no method named `getPay`.

```
7  }
```

```
1  public class SalariedEmployee extends Employee
2  {
3      private double salary; //annual

4      /**
5       Returns the pay for the month.
6      */
7      public double getPay()
8      {
9          return salary/12;
10     }

11     public boolean equals(Object otherObject)
```

<The rest of the definition of `equals` is the same as in the answer to Self-Test Exercise 20 of Chapter 7, but the details of the definition are not important to the current discussion.>

<All constructor and other method definitions are exactly the same as in Display 7.5.>

```
12 }
```

Display 8.6 **Employee Class and Its Derived Classes *(Part 2 of 2)***

```
1  public class HourlyEmployee extends Employee
2  {
3      private double wageRate;
4      private double hours; //for the month

5      /**
6       Returns the pay for the month.
7      */
8      public double getPay()
9      {
10         return wageRate*hours;
11     }

12     public boolean equals(Object otherObject)
13     {
14         if (otherObject == null)
15             return false;
16         else if (getClass() != otherObject.getClass())
17             return false;
18         else
19         {
20             HourlyEmployee otherHourlyEmployee =
21                                     (HourlyEmployee)otherObject;
22              return (super.equals(otherHourlyEmployee)
23                   && (wageRate == otherHourlyEmployee.wageRate)
24                   && (hours == otherHourlyEmployee.hours));
25         }
26     }

   <All constructor and other method definitions are exactly the same as in Display 7.3.>

27 }
```

There is, however, one problem with adding the method `samePay` to the class `Employee`: The method `samePay` includes an invocation of the method `getPay` and the class `Employee` has no `getPay` method. Moreover, there is no reasonable definition we might give for a `getPay` method so that we could add it to the class `Employee`. The only instance variables in the class `Employee` give an employee's name and hire date, but give no information about pay. To see how we should proceed, let's compare objects of the class `Employee` to employees in the real world.

Every real-world employee does have some pay because every real-world employee is either an hourly employee or a salaried employee, and the two derived classes `HourlyEmployee` and `SalariedEmployee` do each have a `getPay` method. The problem is that we do not know how to define the `getPay` method until we know if the employee is an hourly employee or a salaried employee. We would like to postpone the definition of the `getPay` method and only give it in each derived class of the `Employee` class. We would like to simply add a note to the `Employee` class that says: "There will be a method `getPay` for each `Employee` but we do not yet know how it is defined." Java lets us do exactly what we want. The official Java equivalent of our promissory note about the method `getPay` is to make `getPay` an **abstract method**. An abstract method has a heading just like an ordinary method, but no method body. The syntax rules of Java require the modifier `abstract` and require a semicolon in place of the missing method body, as illustrated by the following:

abstract method

```
public abstract double getPay();
```

If we add this abstract method `getPay` to the class `Employee`, then we are free to add the method `samePay` to the class `Employee`.

An abstract method can be thought of as the interface part of a method with the implementation details omitted. Since a private method is normally only a helping method and so not part of the interface for a programmer using the class, it follows that it does not make sense to have a private abstract method. Java enforces this reasoning. In Java, an abstract method cannot be private. Normally an abstract method is public, but protected and package (default) access are allowed.

abstract cannot be private

An abstract method serves a purpose, even though it is not given a full definition. It serves as a placeholder for a method that must be defined in all (nonabstract) derived classes. Note that in Display 8.7 the method `samePay` includes invocations of the method `getPay`. If the abstract method `getPay` were omitted, this invocation of `getPay` would be illegal.

Abstract Method

An abstract method serves as a placeholder for a method that will be fully defined in a descendent class. An abstract method has a complete method heading with the addition of the modifier `abstract`. It has no method body but does end with a semicolon in place of a method body. An abstract method cannot be private.

Examples:

```
public abstract double getPay();

public abstract void doSomething(int count);
```

Display 8.7 Employee Class as an Abstract Class


```
1  /**
2   Class Invariant: All objects have a name string and hire date.
3   A name string of "No name" indicates no real name specified yet.
4   A hire date of Jan 1, 1000 indicates no real hire date specified yet.
5  */
6  public abstract class Employee
7  {
8      private String name;
9      private Date hireDate;

10     public abstract double getPay();

11     public Employee()
12     {
13         name = "No name";
14         hireDate = new Date("Jan", 1, 1000); //Just a placeholder.
15     }

16     public boolean samePay(Employee other)
17     {
18         if (other == null)
19         {
20             System.out.println("Error: null Employee object.");
21             System.exit(0);
22         }
23         //else
24         return (this.getPay() == other.getPay());
25     }

   <All other constructor and other method definitions are exactly the same as in Display 7.2.
                    In particular, they are not abstract methods.>

26 }
```

The class `Date` is defined in Display 4.11, but the details are not relevant to the current discussion of abstract methods and classes. There is no need to review the definition of the class `Date`.

abstract class

A class that has at least one abstract method is called an **abstract class** and, in Java, must have the modifier `abstract` added to the class heading. The redefined, now abstract, class `Employee` is shown in Display 8.7.

An abstract class can have any number of abstract methods. In addition it can have, and typically does have, other regular (fully defined) methods. If a derived class of an abstract class does not give full definitions to all the abstract methods or if the derived class adds an abstract method, then the derived class is also an abstract class and must include the modifier `abstract` in its heading.

concrete class

In contrast with the term abstract class, a class with no abstract methods is called a **concrete class**.

Abstract Class

An **abstract class** is a class with one or more abstract methods. An abstract class must have the modifier `abstract` included in the class heading, as illustrated by the example.

Example:

```
public abstract class Employee
{
    private String name;
    private Date hireDate;

    public abstract double getPay();

        ...
```

Pitfall

You Cannot Create Instances of an Abstract Class

You cannot use an abstract class constructor to create an object of the abstract class. You can only create objects of the derived classes of the abstract class. For example, with the class `Employee` defined as in Display 8.7, the following would be illegal:

```
Employee joe = new Employee(); //Illegal because
                        //Employee is an abstract class.
```

But, this is no problem. The object `joe` could not correspond to any real-world employee. Any real-world employee is either an hourly employee or a salaried employee. In the real world, one cannot be just an employee. One must be either an hourly employee or a salaried employee. Still, it is useful to discuss employees in general. In particular, we can compare employees to see if they have the same pay, even though the way of calculating the pay might be different for the two employees.

Tip

An Abstract Class Is a Type

You cannot create an object of an abstract class (unless it is actually an object of some concrete descendent class). Nonetheless, it makes perfectly good sense to have a parameter of an abstract class type such as `Employee` (as defined in Display 8.7). Then, an object of any of the descendent classes of `Employee` can be plugged in for the parameter. It even makes sense to have a variable of an abstract class type such as `Employee`, although it can only name objects of its concrete descendent classes.

An Abstract Class Is a Type

You can have a parameter of an abstract class type such as the abstract class `Employee` defined in Display 8.7. Then, an object of any of the concrete descendent classes of `Employee` can be plugged in for the parameter. You can also have variables of an abstract class type such as `Employee`, although it can only name objects of its concrete descendent classes.

Self-Test Exercises

10. Can a method definition include an invocation of an abstract method?
11. Can you have a variable whose type is an abstract class?
12. Can you have a parameter whose type is an abstract class?
13. Is it legal to have an abstract class in which all methods are abstract?
14. The abstract class `Employee` (Display 8.7) uses the method definitions from Display 7.2. After we did Display 7.2, we later gave the following improved version of `equals`:

```
public boolean equals(Object otherObject)
{
    if (otherObject == null)
        return false;
    else if (getClass() != otherObject.getClass())
        return false;
    else
    {
        Employee otherEmployee =
                        (Employee)otherObject;
        return (name.equals(otherEmployee.name)
         && hireDate.equals(otherEmployee.hireDate));
    }
}
```

    Would it be legal to replace the version of `equals` for the abstract class `Employee` with this improved version?

15. The abstract class `Employee` given in Display 8.7 has a constructor (in fact, it has more than one, although only one is shown in Display 8.7). But, using a constructor to create an instance of an abstract class, as in the following, is illegal:

```
Employee joe = new Employee(); //Illegal
```

    So, why bother to have any constructors in an abstract class? Aren't they useless?

Chapter Summary

- Late binding (also called dynamic binding) means that the decision of which version of a method is appropriate is decided at run time. Java uses late binding.
- Polymorphism means using the process of late binding to allow different objects to use different method actions for the same method name. *Polymorphism* is essentially another word for late binding.
- You can assign an object of a derived class to a variable of its base class (or any ancestor class), but you cannot do the reverse.
- If you add the modifier `final` to the definition of a method, that indicates that the method may not be redefined in a derived class. If you add the modifier `final` to the definition of a class, that indicates that the class may not be used as a base class to derive other classes.
- An abstract method serves as a placeholder for a method that will be fully defined in a descendent class.
- An abstract class is a class with one or more abstract methods.
- An abstract class is designed to be used as a base class to derive other classes. You cannot create an object of an abstract class type (unless it is an object of some concrete descendent class).
- An abstract class is a type. You can have variables whose type is an abstract class and you can have parameters whose type is an abstract type.

Answers to Self-Test Exercises

1. In essence there is no difference between the two terms. There is only a slight difference in their usage. Late binding refers to the mechanism used to decide which method definition to use when a method is invoked, and polymorphism refers to the fact that the same method name can have different meanings because of late binding.

2. There would be problems well before you wrote the program in Display 8.3. Since `final` means you cannot change the definition of the method `bill` in a derived class, the definition of the method `DiscountSale` would not compile. If you omit the definition of the method `bill` from the class `DiscountSale`, the output would change to

```
floor mat Price and total cost = $10.0
floor mat Price = $11.0 Discount = 10.0%
   Total cost = $11.0
Discounted item is not cheaper.
cup holder Price and total cost = $9.9
cup holder Price = $11.0 Discount = 10.0%
   Total cost = $11.0
Items are not equal.
```

Note that all objects use the definition of `bill` given in the definition of `Sale`.

3. It would not be legal to add it to any class definition because the class `Sale` has no method named `getDiscount` and so the invocation

   ```
   theSale.getDiscount()
   ```

   is not allowed. If the type of the parameter were changed from `Sale` to `DiscountSale`, it would then be legal.

4. ```
   public boolean equals(Object otherObject)
   {
       if (otherObject == null)
           return false;
       else if (getClass() != otherObject.getClass())
           return false;
       else
       {
           DiscountSale otherDiscountSale =
                           (DiscountSale)otherObject;
           return (super.equals(otherDiscountSale)
               && discount == otherDiscountSale.discount);
       }
   }
   ```

5. The definition of `toString` used always depends on the object and not on any type cast. So, the definition used is the same as without the added type cast; that is, the definition of `toString` that is used is the one given in `DiscountSale`.

6. It would not be legal to add it to any class definition because the parameter is of type `Sale` and `Sale` has no method named `getDiscount`. If the parameter type were changed to `DiscountSale`, it would then be legal to add it to any class definition.

7. ```
   ds was changed to map Price $ 5.0 discount 0.0%
   Total cost $5.0
   ```

8. ```
   ds was not changed.
   ```

9. The output would be the following (the main change from Display 8.4 is shown in red):

   ```
   This is the Sale class.
   This is the DiscountSale class.
   No surprises so far, but wait.
   discount2 is a DiscountSale object in a Sale variable.
   Which definition of announcement() will it use?
   This is the DiscountSale class.
   Did it used the Sale version of announcement()?
   ```

10. Yes. See Display 8.7.
   ```

11. Yes, you can have a variable whose type is an abstract class.
12. Yes, you can have a parameter whose type is an abstract class.
13. Yes, it is legal to have an abstract class in which all methods are abstract.
14. Yes, it would be legal to replace the version of `equals` for the abstract class `Employee` with this improved version. In fact, the version of `Employee` on the accompanying CD does use the improved version of `equals`.
15. No, you can still use constructors to hold code that might be useful in derived classes. The constructors in the derived classes can, in fact must, include invocations of constructors in the base (abstract) class. (Recall the use of `super` as a name for the base class constructor.)

Programming Projects

1. Consider a graphics system that has classes for various figures, say rectangles, boxes, triangles, circles, and so on. For example, a rectangle might have data members height, width, and center point, while a box and circle might have only a center point and an edge length or radius, respectively. In a well-designed system these would be derived from a common class, `Figure`. You are to implement such a system.

   The class `Figure` is the base class. You should add only `Rectangle` and `Triangle` classes derived from `Figure`. Each class has stubs for methods `erase` and `draw`. Each of these methods outputs a message telling the name of the class and what method has been called. Since these are just stubs, they do nothing more than output this message. The method `center` calls the `erase` and `draw` methods to erase and redraw the figure at the center. Since you have only stubs for `erase` and `draw`, `center` will not do any "centering" but will call the methods `erase` and `draw`, which will allow you to see which versions of `draw` and `center` it calls. Also, add an output message in the method `center` that announces that `center` is being called. The methods should take no arguments. Also, define a demonstration program for your classes.

   For a real example, you would have to replace the definition of each of these methods with code to do the actual drawing. You will be asked to do this in Programming Project 2.

2. Flesh out Programming Project 1. Give new definitions for the various constructors and methods `center`, `draw`, and `erase` of the class `Figure`; `draw` and `erase` of the class `Triangle`; and `draw` and `erase` of the class `Rectangle`. Use character graphics; that is, the various `draw` methods will place regular keyboard characters on the screen in the desired shape. Use the character `'*'` for all the character graphics. That way the `draw` methods actually draw figures on the screen by placing the character `'*'` at suitable locations on the screen. For the `erase` methods, you can simply clear the screen (by outputting blank lines

or by doing something more sophisticated). There are a lot of details in this project and you will have to decide on some of them on your own.

3. Define a class named `MultiItemSale` that represents a sale of multiple items of type `Sale` given in Display 8.1 (or of the types of any of its descendent classes). The class `MultiItemSale` will have an instance variable whose type is `Sale[]`, which will be used as a partially filled array. There will also be another instance variable of type `int` that keeps track of how much of this array is currently used. The exact details on methods and other instance variables, if any, are up to you. Use this class in a program that obtains information for items of type `Sale` and of type `DiscountSale` (Display 8.2) and computes the total bill for the list of items sold.