CHAPTER 13

Interfaces and Inner Classes

13 Interfaces and Inner Classes

Art, it seems to me, should simplify. That, indeed, is very nearly the whole of the higher artistic process; finding what conventions of form and what details one can do without and yet preserve the spirit of the whole...

Willa Sibert Cather, *On the Art of Fiction*

Introduction

A Java *interface* specifies a set of methods that any class that implements the interface must have. An interface is itself a type, which allows you to define methods with parameters of an interface type and then have the code apply to all classes that implement the interface. One way to view an interface is as an extreme form of an abstract class. However, as you will see, an interface allows you to do more than an abstract class allows you to do. Interfaces are Java's way of approximating multiple inheritance. You cannot have multiple base classes in Java, but interfaces allow you to approximate the power of multiple base classes.

The second major topic of this chapter is *inner classes.* An inner class is simply a class defined within another class. Since inner classes are local to the class that contains them, they can help make a class self-contained by allowing you to make helping classes inner classes.

Prerequisites

Section 13.1 on interfaces and Section 13.2 on simple uses of inner classes are independent of each other and can be covered in any order. Section 13.3 on more subtle details of inner classes requires both Sections 13.1 and 13.2.

Section 13.1 on interfaces requires Chapters 1 through 9. No material from Chapters 10 through 12 is used anywhere in this chapter.

Section 13.2 on simple uses of inner classes requires Chapters 1 through 5. It does not use any material from Chapters 6 through 12.

Section 13.3 on more advanced inner class material requires both Sections 13.1 and 13.2 (and of course their prerequisites). The material in Section 13.3 is not used elsewhere in this book.

13.1 Interfaces

Autonomy of Syntax

A linguistic concept attributed to Noam Chomsky

In this section we describe *interfaces*. An interface is a type that groups together a number of different classes that all include method definitions for a common set of method headings.

INTERFACES

An **interface** is something like the extreme case of an abstract class. *An interface is not a class. It is, however, a type that can be satisfied by any class that implements the interface.* An interface is a property of a class that says what methods it must have. interface

An interface specifies the headings for methods that must be defined in any class that implements the interface. For example, Display 13.1 shows an interface named `Ordered`. Note that an interface contains only method headings. It contains no instance variables nor any complete method definitions. (Although, as we will see, it can contain defined constants.)

To **implement an interface**, a concrete class (that is, a class other than an abstract class) must do two things: implementing an interface

1. It must include the phrase

   ```
   implements Interface_Name
   ```

   at the start of the class definition. To implement more than one interface, you list all the interface names, separated by commas, as in

   ```
   implements SomeInterface, AnotherInterface
   ```

Display 13.1 The Ordered Interface


```
1  public interface Ordered
2  {
3      public boolean precedes(Object other);

4      /**
5       For objects of the class o1 and o2,
6       o1.follows(o2) == o2.preceded(o1).
7      */
8      public boolean follows(Object other);
9  }
```

Do not forget the semicolons at the end of the method headings.

Neither the compiler nor the run-time system will do anything to ensure that this comment is satisfied. It is only advisory to the programmer implementing the interface.

2. The class must implement *all* the method headings listed in the definition(s) of the interface(s).

For example, to implement the `Ordered` interface, a class definition must contain the phrase `implements Ordered` at the start of the class definition, as shown in the following:

```
public class OrderedHourlyEmployee
        extends HourlyEmployee implements Ordered
{
```

The class must also implement the two methods `precedes` and `follows`. The full definition of `OrderedHourlyEmployee` is given in Display 13.2.

Display 13.2 Implementation of an Interface


```
1  public class OrderedHourlyEmployee
2          extends HourlyEmployee implements Ordered
3  {
4      public boolean precedes(Object other)
5      {
6          if (other == null)
7              return false;
8          else if (!(other instanceof HourlyEmployee))
9              return false;
10         else
11         {
12             OrderedHourlyEmployee otherOrderedHourlyEmployee =
13                                 (OrderedHourlyEmployee)other;
14              return (getPay() < otherOrderedHourlyEmployee.getPay());
15         }
16     }

17     public boolean follows(Object other)
18     {
19         if (other == null)
20             return false;
21         else if (!(other instanceof OrderedHourlyEmployee))
22             return false;
23         else
24         {
25             OrderedHourlyEmployee otherOrderedHourlyEmployee =
26                                 (OrderedHourlyEmployee)other;
27             return (otherOrderedHourlyEmployee.precedes(this));
28         }
29     }
30 }
```

Although `getClass` works better than `instanceof` for defining `equals`, `instanceof` works better here. However, either will do for the points being made here.

An interface and all of its method headings are normally declared to be public. They cannot be given private, protected, or package access. (The modifier `public` may be omitted, but all the methods will still be treated as if they are public.) When a class implements an interface, it must make all the methods in the interface public.

An interface is a type. This allows you to write a method with a parameter of an interface type, such as a parameter of type `Ordered`, and that parameter will accept as an argument any class you later define that implements the interface.

An interface serves a function similar to a base class, but it is important to note that it is not a base class. (In fact, it is not a class of any kind.) Some programming languages (such as C++) allow one class to be a derived class of two or more different base classes. This is not allowed in Java. In Java, a derived class can have only one base class. However, in addition to any base class that a Java class may have, it can also implement any number of interfaces. This allows Java programs to approximate the power of multiple base classes without the complications that can arise with multiple base classes.

You might want to say the argument to `precedes` in the `Ordered` interface (Display 13.2) is the same as the class doing the implementation (for example, `OrderedHourly-Employee`). There is no way to say this in Java, so we normally make such parameters of type `Object`. It would be legal to make the argument to `precedes` of type `Ordered`, but that is not normally preferable to using `Object` as the parameter type. If you make the argument of type `Ordered`, you would still have to handle the case of `null` and the case of an argument that (while `Ordered`) is not of type `OrderedHourlyEmployee`.

An interface definition is stored in a `.java` file and compiled just as a class definition is compiled.

ABSTRACT CLASSES IMPLEMENTING INTERFACES

As you saw in the previous subsection, a concrete class (that is, a regular class) must give definitions for all the method headings given in an interface in order to implement the interface. However, you can define an abstract class that implements an interface but only gives definitions for some of the method headings given in the interface. The method headings given in the interface that are not given definitions are made into abstract methods. A simple example is given in Display 13.3.

DERIVED INTERFACES

You can derive an interface from a base interface. This is often called **extending** the interface. The details are similar to deriving a class. An example is given in Display 13.4.

extending an interface

Self-Test Exercises

1. Can you have a variable of an interface type? Can you have a parameter of an interface type?
2. Can an abstract class ever implement an interface?

Display 13.3 An Abstract Class Implementing an Interface


```
1  public abstract class MyAbstractClass implements Ordered
2  {
3      int number;
4      char grade;
5
6      public boolean precedes(Object other)
7      {
8          if (other == null)
9              return false;
10         else if (!(other instanceof HourlyEmployee))
11             return false;
12         else
13         {
14             MyAbstractClass otherOfMyAbstractClass =
15                                             (MyAbstractClass)other;
16             return (this.number < otherOfMyAbstractClass.number);
17         }
18     }

19     public abstract boolean follows(Object other);

20 }
```

Display 13.4 Extending an Interface


```
1  public interface ShowablyOrdered extends Ordered
2  {
3      /**
4        Outputs an object of the class that precedes the calling object.
5      */
6      public void showOneWhoPrecedes();
7  }
```

Neither the compiler nor the run-time system will do anything to ensure that this comment is satisfied.

A (concrete) class that implements the `ShowablyOrdered` interface must have a definition for the method `showOneWhoPrecedes` and also have definitions for the methods `precedes` and `follows` given in the `Ordered` interface.

3. Can a derived class have two base classes? Can it implement two interfaces?
4. Can an interface implement another interface?

Interfaces

An interface is a type that specifies method headings (and, as we will see, possibly defined constants as well). The syntax for defining an interface is similar to the syntax of defining a class, except that the word `interface` is used in place of `class` and only the method headings without any method body (but followed by a semicolon) are given.

Note that an interface has no instance variables and no method definitions.

A class can implement any number of interfaces. To implement an interface the class must include

```
implements Interface_Name
```

at the end of the class heading and must supply definitions for the method headings given in the interface. If the class does not supply definitions for all the method headings given in the interface, then the class must be an abstract class and the method headings without definitions must be abstract methods.

Example:

See Displays 13.1, 13.2, and 13.3.

Pitfall

Interface Semantics Are Not Enforced

As far as the Java compiler is concerned, an interface has syntax but no semantics. For example, the definition of the `Ordered` interface (Display 13.1) says the following in a comment:

```
/**
 For objects of the class o1 and o2,
 o1.follows(o2) == o2.preceded(o1).
*/
```

You might have assumed that this is true even if there were no comment in the interface. After all, in the real world, if I precede you, then you follow me. However, that is giving your intuitive interpretation to the word "`precedes`."

As far as the compiler and run-time systems are concerned, the `Ordered` interface merely says that the methods `precedes` and `follows` each take one argument of type `Object` and return a `boolean` value. The interface does not really require that the `boolean` value be computed in any particular way. For example, the compiler would be satisfied if both `precedes` and `follows` always return `true` or if they always return `false`. It would even allow the methods to use a random number generator to generate a random choice between `true` and `false`.

It would be nice if we could safely give an interface some simple semantics, such as saying that `o1.follows(o2)` means the same as `o2.preceded(o1)`. However, if Java did allow that, there would be problems with having a class implement two interfaces or even with having a class

derived from one base class and implementing an interface. Either of these situations could produce two semantic conditions both of which must be implemented for the same method, and the two semantics may not be consistent. For example, suppose that (contrary to fact) you could require that `o1.follows(o2)` means the same as `o2.preceded(o1)`. You could also define another interface with an inconsistent semantics, such as saying that `precedes` always returns `true` and that `follows` always returns `false`. As long as a class can have two objects, there is no way a class could implement both of these semantics. Interfaces in Java are very well behaved, the price of which is that you cannot count on Java to enforce any semantics in an interface.

If you want to require semantics for an interface, you can add it to the documentation, as illustrated by the comment in Display 13.1 and the comment in Display 13.4, but always remember that these are just comments; they are not enforced by either the compiler or the run-time system, so you cannot necessarily rely on such semantics being followed. However, we live in an imperfect world, and sometimes you will find that you must specify a semantics for an interface; you do so in the interface's documentation. It then becomes the responsibility of the programmers implementing the interface to follow the semantics.

Having made our point about interface semantics not being enforced by the compiler or run-time system, we want to nevertheless urge you to follow the specified semantics for an interface. Software written for classes that implement an interface will assume that any class that implements the interface does satisfy the specified semantics. So, if you define a class that implements an interface but does not satisfy the semantics for the interface, then software written for classes that implement that interface will probably not work correctly for your class.

Interface Semantics Are Not Enforced

When you define a class that implements an interface, the compiler and run-time system will let you define the body of an interface method any way you want, provided you keep the method heading as it is given in the interface. However, you should follow the specified semantics for an interface whenever you define a class that implements that interface; otherwise, software written for that interface may not work for your class.

THE `Comparable` INTERFACE

This subsection requires material on arrays from Chapter 6. If you have not yet covered Chapter 6, you can skip this section and the following Programming Example without any loss of continuity. If you have read Chapter 6, you should not consider this section to be optional.

In Chapter 6 (Display 6.9) we gave a method for sorting a partially filled array of base type `double` into increasing order. It is very easy to transform the code into a method to sort into decreasing order instead of increasing order. (See Self-Test Exercise 18 of Chapter 6 and its answer if this is not clear to you.) It is also easy to modify the

code to obtain methods for sorting integers instead of doubles or sorting strings into alphabetical order. Although these changes are easy, they seem to be, and in fact are, a useless nuisance. All these sorting methods will be essentially the same. The only differences are the types of the values being sorted and the definition of the ordering. It would seem that we should be able to give a single sorting method that covers all these cases. The Comparable interface lets us do this.

The Comparable interface is in the java.lang package and so is automatically available to your program. The Comparable interface has only the following method heading that must be implemented for a class to implement the Comparable interface: compareTo

```
public int compareTo(Object other);
```

The Comparable interface has a semantics, and it is the programmer's responsibility to follow this semantics when implementing the Comparable interface. The semantics says that compareTo returns

a negative number if the calling object "comes before" the parameter other,

a zero if the calling object "equals" the parameter other,

and a positive number if the calling object "comes after" the parameter other.[1]

Almost any reasonable notions of "comes before" should be acceptable. In particular, all of the standard less-than relations on numbers and lexicographic ordering on strings are suitable ordering for compareTo. (The relationship "comes after" is just the reverse of "comes before.") If you need to consider other ordering, the precise rule is that the ordering must be a total ordering, which means the following rules must be satisfied:

(Irreflexive) For no object o does o come before o.

(Trichotomy) For any two objects o1 and o2, one and only one of the following holds true: o1 comes before o2, o1 comes after o2, or o1 equals o2.

(Transitivity) If o1 comes before o2 and o2 comes before o3, then o1 comes before o3.

The "equals" of the compareTo method semantics should coincide with the equals methods if that is possible, but this is not absolutely required by the semantics.

If you define a class that implements the Comparable interface but that does not satisfy these conditions, then code written for Comparable objects will not work properly. It is the responsibility of you the programmer to ensure that the semantics is satisfied. Neither the compiler nor the run-time system enforces any semantics on the Comparable interface.

If you have read this subsection, you should also read the following Programming Example.

---

[1] Since the parameter to CompareTo is of type Object, an argument to CompareTo might not be an object of the class being defined. If the parameter other is not of the same type as the class being defined, then the semantics specifies that a ClassCastException should be thrown.

The Comparable Interface

The `Comparable` interface is in the `java.lang` package and so is automatically available to your program. The `Comparable` interface has only the following method heading that must be given a definition for a class to implement the `Comparable` interface:

```
public int compareTo(Object other);
```

The method `compareTo` should return

- a negative number if the calling object "comes before" the parameter `other`,
- a zero if the calling object "equals" the parameter `other`,
- and a positive number if the calling object "comes after" the parameter `other`.

The "comes before" ordering that underlies `compareTo` should be a total ordering. Most normal ordering, such as less-than ordering on numbers and lexicographic ordering on strings, are total ordering.

Example

Using the Comparable Interface

Comparable

Display 13.5 shows a class with a method that can sort any partially filled array whose base type implements the `Comparable` interface (including implementing the semantics we discussed in the previous subsection). To obtain the code in Display 13.5 we started with the sorting code in Display 6.9 and mechanically replaced all occurrences of the array type `double[]` with the type `Comparable[]` and we replaced all Boolean expressions of the form

Expression_1 < *Expression_2*

with

Expression_1`.compareTo(`*Expression_2*`) < 0`

We also changed the comments a bit to make them consistent with the `compareTo` notation. The changes are highlighted in Display 13.5. Only four small changes to the code were needed.

Display 13.6 shows a demonstration of using the sorting method given in Display 13.5. To understand why the demonstration works, you need to be aware of the fact that both of the classes `Double` and `String` implement the `Comparable` interface.

If you were to check the full documentation for the class `Double` you would see that `Double` implements the `Comparable` interface and so has a `compareTo` method. Moreover, for objects `o1` and `o2` of `Double`,

```
o1.compareTo(o2) < 0 //o1 "comes before" o2
```

Display 13.5 Sorting Method for Array of Comparable *(Part 1 of 2)*

```
1  public class GeneralizedSelectionSort
2  {
3      /**
4       Precondition: numberUsed <= a.length;
5                     The first numberUsed indexed variables have values.
6       Action: Sorts a so that a[0, a[1], ... , a[numberUsed – 1] are in
7       increasing order by the compareTo method.
8      */
9      public static void sort(Comparable[] a, int numberUsed)
10     {
11         int index, indexOfNextSmallest;
12         for (index = 0; index < numberUsed – 1; index++)
13         {//Place the correct value in a[index]:
14             indexOfNextSmallest = indexOfSmallest(index, a, numberUsed);
15             interchange(index,indexOfNextSmallest, a);
16              //a[0], a[1],..., a[index] are correctly ordered and these are
17              //the smallest of the original array elements. The remaining
18              //positions contain the rest of the original array elements.
19         }
20     }

21     /**
22      Returns the index of the smallest value among
23      a[startIndex], a[startIndex+1], ... a[numberUsed – 1]
24     */
25     private static int indexOfSmallest(int startIndex,
26                                        Comparable[] a, int numberUsed)
27     {
28         Comparable min = a[startIndex];
29         int indexOfMin = startIndex;
30         int index;
31         for (index = startIndex + 1; index < numberUsed; index++)
32             if (a[index].compareTo(min) < 0)//if a[index] is less than min
33             {
34                 min = a[index];
35                 indexOfMin = index;
36                 //min is smallest of a[startIndex] through a[index]
37             }
38         return indexOfMin;
39     }
```

Display 13.5 Sorting Method for Array of Comparable *(Part 2 of 2)*

```
    /**
     Precondition: i and j are legal indices for the array a.
     Postcondition: Values of a[i] and a[j] have been interchanged.
    */
    private static void interchange(int i, int j, Comparable[] a)
    {
        Comparable temp;
        temp = a[i];
        a[i] = a[j];
        a[j] = temp; //original value of a[i]
    }

}
```

Display 13.6 Sorting Arrays of Comparable *(Part 1 of 2)*


```
1  /**
2   Demonstrates sorting arrays for classes that
3   implement the Comparable interface.
4  */
5  public class ComparableDemo
6  {
7      public static void main(String[] args)
8      {
9          Double[] d = new Double[10];
10         for (int i = 0; i < d.length; i++)
11             d[i] = new Double(d.length - i);

12         System.out.println("Before sorting:");
13         int i;
14         for (i = 0; i < d.length; i++)
15             System.out.print(d[i].doubleValue() + ", ");
16         System.out.println();

17         GeneralizedSelectionSort.sort(d, d.length);

18         System.out.println("After sorting:");
19         for (i = 0; i < d.length; i++)
20             System.out.print(d[i].doubleValue() + ", ");
21         System.out.println();
```

The classes `Double` and `String` do implement the `Comparable` interface.

Display 13.6 Sorting Arrays of Comparable *(Part 2 of 2)*

```
22          String[] a = new String[10];
23          a[0] = "dog";
24          a[1] = "cat";
25          a[2] = "cornish game hen";
26          int numberUsed = 3;

27          System.out.println("Before sorting:");
28          for (i = 0; i < numberUsed; i++)
29              System.out.print(a[i] + ", ");
30          System.out.println();
31
32          GeneralizedSelectionSort.sort(a, numberUsed);

33          System.out.println("After sorting:");
34          for (i = 0; i < numberUsed; i++)
35              System.out.print(a[i] + ", ");
36          System.out.println();
37      }
38  }
```

SAMPLE DIALOGUE

```
Before Sorting
10.0, 9.0, 8.0, 7.0, 6.0, 5.0, 4.0, 3.0, 2.0, 1.0,
After sorting:
1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0,
Before sorting;
dog, cat, cornish game hen,
After sorting:
cat, cornish game hen, dog,
```

means the same thing as

```
o1.doubleValue() < o2.doubleValue()
```

So, the implementation of the `Comparable` interface for the class `Double` is really just the ordinary less-than relationship on the `double` values corresponding to the `Double` objects.

Similarly, if you were to check the full documentation for the class `String`, you would see that `String` implements the `Comparable` interface and so has a `compareTo` method. Moreover, the implementation of the `compareTo` method for the class `String` is really just the ordinary lexicographic relationship on the strings.

This Programming Example used the standard library classes `Double` and `String` for the base type of the array. You can do the same thing with arrays whose base class is a class you defined, so long as the class implements the `Comparable` interface (including the standard semantics, which we discussed earlier).

This Programming Example does point out one restriction on interfaces. They can only apply to classes. A primitive type cannot implement an interface. So, in Display 13.6 we could not sort an array with base type `double` using the sorting method for an array of `Comparable`. We had to settle for sorting an array with base type `Double`. This is a good example of using a wrapper class with its "wrapper class personality."

Self-Test Exercises

These exercises are for the material on the `Comparable` interface.

5. The method `interchange` in Display 13.5 makes no use of the fact that its second argument is an array with base type `Comparable`. Suppose we change the parameter type `Comparable[]` to `Object[]` and change the type of the variable `temp` to `Object`. Would the program in Display 13.6 produce the same dialog?

6. Is the following a suitable implementation of the `Comparable` interface?

```
public class Double2 implements Comparable
{
    private double value;

    public Double2(double theValue)
    {
        value = theValue;
    }

    public int compareTo(Object other)
    {
        return –1;
    }

    public double doubleValue()
    {
        return value;
    }
}
```

You can think of the underlying "comes before" relationship as saying that for any objects `d1` and `d2`, `d1` comes before `d2`.

7. Suppose you have a class `Circle` that represents circles all of whose centers are at the same point. (To make it concrete you can take the circles to be in the usual *x*,*y* plain and to all have their centers at the origin.) Suppose there is a `boolean` valued method `inside` of the class `Circle` such that, for circles `c1` and `c2`,

   `c1.inside(c2)`

   returns `true` if `c1` is completely inside of `c2` (and `c2` is not the same as `c1`). Is the following a total ordering?

   `c1` comes before `c2` if `c1` is inside of `c2`
   (that is, if `c1.inside(c2)` returns `true`).

   You could represent objects of the class `Circle` by a single value of type `double` that gives the radius of the circle, but the answer does not depend on such details.

DEFINED CONSTANTS IN INTERFACES

The designers of Java often used the interface mechanism to take care of a number of miscellaneous details that do not really fit the spirit of what an interface is supposed to be. One example of this is the use of an interface as a way to name a group of defined constants.

An interface can contain defined constants as well as method headings or instead of method headings. When a method implements the interface, it automatically gets the defined constants. For example, the following interface defines constants for months:

```
public interface MonthNumbers
{
    public static final int JANUARY = 1,
       FEBRUARY = 2, MARCH = 3, APRIL = 4, MAY = 5,
       JUNE = 6, JULY = 7, AUGUST = 8, SEPTEMBER = 9,
       OCTOBER = 10, NOVERMBER = 11, DECEMBER = 12;
}
```

Any class that implements the `MonthNumbers` interface will automatically have the 12 constants defined in the `MonthNumbers` interface. For example, consider the following toy class:

```
public class DemoMonthNumbers implements MonthNumbers
{
    public static void main(String[] args)
    {
        System.out.println(
           "The number for January is " + JANUARY);
    }
}
```

Note that the constant `JANUARY` is used in the class `DemoMonthNumbers` but is not defined there. The class `DemoMonthNumbers` automatically gets the month constants because it implements the `MonthNumbers` interface.

no instance variables

An interface cannot have instance variables, although it can use the syntax for instance variables as a way to define constants. Any variables defined in an interface must be public, static, and final, so Java allows you to omit those modifiers. The following is an equivalent definition of the interface `MonthNumbers`:

```
public interface MonthNumbers
{
    int JANUARY = 1,
       FEBRUARY = 2, MARCH = 3, APRIL = 4, MAY = 5,
       JUNE = 6, JULY = 7, AUGUST = 8, SEPTEMBER = 9,
       OCTOBER = 10, NOVERMBER = 11, DECEMBER = 12;
}
```

Thus, an interface can be used to give a name for a group of defined constants, so that you can easily add the needed constants to any class by implementing the interface. This is really a different use for interfaces than what we have seen before, which was to use interfaces to specify method headings. It is legal to mix these two uses by including both defined constants and method headings in a single interface.

Pitfall

Inconsistent Interfaces

Java allows a class to have only one base class but also allows it to implement any number of interfaces. The reason that a class can have only one base class is that if Java allowed two base classes, the two base classes could provide different and inconsistent definitions of a single method heading. Since interfaces have no method bodies at all, this problem cannot arise when a class implements two interfaces. The ideal that the designers of Java apparently hoped to realize was that any two interfaces will always be consistent. However, this ideal was not fully realized. Although it is a rare phenomenon, two interfaces can be inconsistent. In fact, there is more than one kind of inconsistency that can be exhibited. If you write a class definition that implements two inconsistent interfaces, that is an error and the class definition is illegal. Let's see how two interfaces can be inconsistent.

inconsistent constants

The most obvious way that two interfaces can be inconsistent is by defining two constants with the same name but with different values. For example:

```
public interface Interface1
{
    int ANSWER = 42;
}

public interface Interface2
```

```
{
    int ANSWER = 0;
}
```

Suppose a class definition begins with

```
public class MyClass
          implements Interface1, Interface2
{ ...
```

Clearly this has to be, and is, illegal. The defined constant ANSWER cannot be simultaneously 42 and 0.[2]

inconsistent method headings

Even two method headings can be inconsistent. For example, consider the following two interfaces:

```
public interface InterfaceA
{
    public int getStuff();
}

public interface InterfaceB
{
    public String getStuff();
}
```

Suppose a class definition begins with

```
public class YourClass
               implements InterfaceA, InterfaceB
{ ...
```

Clearly this has to be, and is, illegal. The method `getStuff` in the class `YourClass` cannot be simultaneously a method that returns an `int` and a method that returns a value of type `String`. (Remember that you cannot overload a method based on the type returned; so, overloading cannot be used to get around this problem.)

Self-Test Exercises

8. Will the following program compile? If it does compile will it run? `Interface1` and `Interface2` were defined in the previous subsection.

```
public class MyClass
             implements Interface1, Interface2
{
```

[2] If the class never uses the constant ANSWER, then there is no inconsistency and the class will compile and run with no error messages.

```
    public static void main(String[] args)
    {
        System.out.println(ANSWER);
    }
}
```

9. Will the following program compile? If it does compile will it run? `Interface1` and `Interface2` were defined in the previous subsection.

```
public class MyClass
             implements Interface1, Interface2
{
    public static void main(String[] args)
    {
        System.out.println("Hello");
    }
}
```

10. Will the following program compile? If it does compile will it run? `InterfaceA` and `InterfaceB` were defined in the previous subsection.

```
public class YourClass
               implements InterfaceA, InterfaceB
{
    public String getStuff()
    {
        return "one";
    }
}
```

11. Will the following two interfaces and the following program class compile? If they compile will the program run with no error messages?

```
public interface InterfaceA
{
    public int getStuff();
}

public interface InterfaceOtherB
{
    public String getStuff(String someStuff);

}

public class OurClass
            implements InterfaceA, InterfaceOtherB
{
```

```
    private int intStuff = 42;

    public static void main(String[] args)
    {
        OurClass object = new OurClass();
        System.out.println(object.getStuff()
                    + object.getStuff(" Hello"));
    }

    public int getStuff()
    {
        return intStuff;
    }

    public String getStuff(String someStuff)
    {
        return someStuff;
    }
}
```

THE `Serializable` INTERFACE

As we have already noted, the designers of Java often used the interface mechanism to take care of miscellaneous details that do not really fit the spirit of what an interface is supposed to be. An extreme example of this is the `Serializable` interface. The `Serializable` interface has no method headings and no defined constants. As a traditional interface it is pointless. However, Java uses it as a type tag that means the programmer gives permission to the system to implement file I/O in a particular way. If you want to know what that way of implementing file I/O is, see Chapter 10, in which the `Serializable` interface is discussed in detail.

`Serializable`

THE `Cloneable` INTERFACE

The `Cloneable` interface is another example where Java uses the interface mechanism for something other than its traditional role. The `Cloneable` interface has no method headings that must be implemented (and has no defined constants). However, it is used to say something about how the method `clone`, which is inherited from the class `Object`, should be used and how it should be redefined.

`Cloneable`

So, what is the purpose of the `Cloneable` interface? When you define a class to implement the `Cloneable` interface, you are agreeing to redefine the `clone` method (inherited from `Object`) in a particular way. The primary motivation for this appears to be security issues. Cloning can potentially copy supposedly private data if not done correctly. Also, some software may depend on your redefining the `clone` method in a certain way. Programmers have strong and differing views on how to handle cloning and the `Cloneable` interface. What follows is the official Java line on how to do it.

The method `Object.clone()` does a bit-by-bit copy of the object's data in storage. If the data is all primitive type data or data of immutable class types, such as `String`, then this works fine and has no unintended side effects. However, if the data in the object includes instance variables whose type is a mutable class, then this would cause what we refer to as *privacy leaks*. (See the Pitfall section entitled "Privacy Leaks" in Chapter 5.) So, when implementing the `Cloneable` interface for a class, you should invoke the `clone` method of the base class `Object` (or whatever the base class is) and then change the new instance variables whose type is a mutable class. There are also issues of exception handling to deal with. An example may be clearer than an abstract discussion.

Let's start with the simple case. Suppose your class has no instance variables of a mutable class type, or to phrase it differently, suppose your class has instance variables all of whose types are either a primitive type or an immutable class type, like `String`. And to make it even simpler, suppose your class has no specified base class, so the base class is `Object`. If you want to implement the `Cloneable` interface, you should define the `clone` method as in Display 13.7.

The `try-catch` blocks are required because the inherited method `clone` can throw the exception `CloneNotSupportedException` if the class does not implement the `Cloneable` interface. Of course, in this case the exception will never be thrown, but the compiler will still insist on the `try-catch` blocks.

Now let's suppose your class has one instance variable of a mutable class type named `DataClass`. Then, the definition of the `clone` method should be as in Display 13.8.

Display 13.7 Implementation of the Method `clone` (Simple Case)


```
1  public class YourCloneableClass implements Cloneable
2  {                                  Works correctly if each instance variable is of a
3          .                          primitive type or of an immutable type like String.
4          .
5          .
6      public Object clone()
7      {
8         try
9         {
10           return super.clone();//Invocation of clone
11                                 //in the base class Object
12        }
13        catch(CloneNotSupportedException e)
14        {//This should not happen.
15           return null; //To keep the compiler happy.
16        }
17     }
18         .
19         .
20         .
21 }
```

Display 13.8 Implementation of the Method clone (Harder Case)


```
1  public class YourCloneableClass2 implements Cloneable
2  {
3      private DataClass someVariable;
4          .
5          .
6          .
7      public Object clone()
8      {
9          try
10         {
11             YourCloneableClass2 copy =
12                                 (YourCloneableClass2)super.clone();
13             copy.someVariable = (DataClass)someVariable.clone();
14             return copy;
15         }
16         catch(CloneNotSupportedException e)
17         {//This should not happen.
18             return null; //To keep the compiler happy.
19         }
20     }
21         .
22         .
23         .
24 }
```

DataClass is a mutable class. Any other instance variables are each of a primitive type or of an immutable type like String.

The class DataClass must also properly implement the Cloneable interface including defining the clone method as we are describing.

First a bit-by-bit copy of the object is made by the invocation of `super.clone()`. The dangerous part of copy is the reference to the mutable object in the instance variable `someVariable`. So, the reference is replaced by a reference to a copy of the object named by `someVariable`. This is done with the line

```
copy.someVariable = (DataClass)someVariable.clone();
```

The object named by copy is now safe and so can be returned by the `clone` method.

If there are more instance variables that have a mutable class type, then you repeat what we did for `someVariable` for each of the mutable instance variables.

This requires that any mutable class type for an instance variable, such as the class `DataClass`, also correctly implement the `Cloneable` interface. So, the definition of `DataClass` should also follow the model of Display 13.7 or Display 13.8, whichever is appropriate. Similarly, any mutable class for instance variables in `DataClass` must properly implement the `Cloneable` interface, and so forth.

The designers of Java did make some effort to force you to implement the `Cloneable` interface for instance variable classes like `DataClass` in Display 13.8. The method `clone` of the class `Object` is marked `protected`. If you do not override the `clone`

method in `DataClass` so that `clone` is `public`, then the code in Display 13.8 will give a compiler error message saying that the following is illegal because `DataClass.clone` is protected (as inherited from `Object`):

```
copy.someVariable = (DataClass)someVariable.clone();
```

This is some check but not a strong check. If you do not implement the `Cloneable` interface for `DataClass` but you define `clone` in `DataClass` in any way at all, so long as it is `public`, then you will not get this error message. You the programmer have the responsibility to implement the `Cloneable` interface as specified in the Java documentation, but the enforcement of this is spotty.

The same basic technique applies if your class is derived from some class other than `Object`, except that, in this case, there normally is no required exception handling. To implement the `Cloneable` interface in a derived class with a base class other than `Object`, the details are as follows: The base class must properly implement the `Cloneable` interface, and the derived class must take care of any mutable class instance variable added in the definition of the derived class. These new mutable class instance variables are handled by the technique shown in Display 13.8 for the instance variable `someVariable`. As long as the base class properly implements the `Cloneable` interface, including defining the `clone` method as we are describing, then the derived class's `clone` method need not worry about any inherited instance variables. Usually, you need not have `try` and `catch` blocks for `CloneNotSupportedException` because the base class `clone` method, `super.clone()`, normally catches all its `CloneNotSupportedException`s and so `super.clone()` will never throw a `CloneNotSupportedException`. (See Self-Test Exercise 13 for an example.)

Self-Test Exercises

12. Modify the following class definition so it correctly implements the `Cloneable` interface (all the instance variables are shown):

```
public class StockItem
{
    private int number;
    private String name;
    public void setNumber(int newNumber)
    {
        number = newNumber;
    }
       ...
}
```

13. Modify the following class definition so it correctly implements the `Cloneable` interface (all the new instance variables are shown):

```
public class PricedItem extends StockItem
{
```

```
    private double price;

        ...
}
```

14. Modify the following class definition so it correctly implements the `Cloneable` interface (all the instance variables are shown):

```
public class Record
{
    private StockItem item1;
    private StockItem item2;
    private String description;

        ...
}
```

15. Modify the following class definition so it correctly implements the `Cloneable` interface (all the new instance variables are shown):

```
public class BigRecord extends Record
{
    private StockItem item3;

        ...
}
```

16. Modify the definition of the class `Date` (Display 4.11) so it implements the `Cloneable` interface. Be sure to define the method `clone` in the style of Display 13.7.
17. Modify the definition of the class `Employee` (Display 7.2) so it implements the `Cloneable` interface. Be sure to define the method `clone` in the style of Display 13.8.
18. Modify the definition of the class `HourlyEmployee` (Display 7.3) so it implements the `Cloneable` interface. Be sure to define the method `clone` in the style of Display 13.8.

13.2 Simple Uses of Inner Classes

The ruling ideas of each age have ever been the ideas of its ruling class.

Karl Marx and Friedrich Engels, *The Communist Manifesto*

Inner classes are classes defined within other classes. In this section, we will describe one of the most useful applications of inner classes, namely, inner classes used as helping classes.

HELPING CLASSES

inner class

Defining an **inner class** is straightforward; simply include the definition of the inner class within another class, as follows:

```
public class OuterClass
{
    private class InnerClass
    {
        Declarations_of_InnerClass_Instance_Variables
        Definitions_of_InnerClass_Methods
    }

    Declarations_of_OuterClass_Instance_Variables
    Definitions_of_OuterClass_Methods
}
```

outer class

As this outline suggests, the class that includes the inner class is called an **outer class**. The definition of the inner class (or classes) need not be the first item(s) of the outer class, but it is good to place it either first or last so that it is easy to find. The inner class need not be private, but that is the only case we will consider in this section. We will consider other modifiers besides `private` in Section 13.3.

An inner class definition is a member of the outer class in the same way that the instance variables of the outer class and the methods of the outer class are members of the outer class. Thus, an inner class definition is local to the outer class definition. So you may reuse the name of the inner class for something else outside the definition of the outer class. If the inner class is private, as ours will always be in this section, then the inner class cannot be accessed by name outside the definition of the outer class.

There are two big advantages to inner classes. First, because they are defined within a class, they can be used to make the outer class self-contained or more self-contained than it would otherwise be. The second advantage is that the inner and outer classes' methods have access to each other's private methods and private instance variables.

Tip

INNER AND OUTER CLASSES HAVE ACCESS TO EACH OTHER'S PRIVATE MEMBERS

Within the definition of a method of an inner class, it is legal to reference a private instance variable of the outer class and to invoke a private method of the outer class. To facilitate this, Java follows this convention: If a method is invoked in an inner class and the inner class has no such method, then it is assumed to be an invocation of the method by that name in the outer class. (If the outer class also has no method by that name, that is, of course, an error.) Similarly, an inner class can use the name of an instance variable of the outer class.

The reverse situation, invoking a method of the inner class from the outer class, is not so simple. To invoke a (nonstatic) method of the inner class from within a method of the outer class, you need an object of the inner class to use as a calling object, as we did in Display 13.9 (Part 2).

As long as you are within the definition of the inner or outer classes, the modifiers `public` and `private` (used within the inner or outer classes) are equivalent.

These sorts of invocations and variable references that cross between inner and outer classes can get confusing. So, it is best to confine such invocations and variable references to cases that are clear and straightforward. It is easy to tie your code in knots if you get carried away with this sort of thing.

Access Privileges between Inner and Outer Classes

Inner and outer classes have access to each other's private members.

Example

A Bank Account Class

Display 13.9 contains a simplified bank account program with an inner class for amounts of money. The bank account class uses values of type `String` to obtain or return amounts of money, such as the amount of a deposit or the answer to a query for the account balance. However, inside the class it stores amounts of money as values of type `Money`, which is an inner class. Values of type `Money` are not stored as `String`s, which would be difficult to do arithmetic on, nor are they stored as values of type `double`, which would allow round-off errors that would not be acceptable in banking transactions. Instead, the class `Money` stores amounts of money as two integers, one for the dollars and one for the cents. In a real banking program, the class `Money` might have a larger collection of methods, such as methods to do addition, subtraction, and compute percentages, but in this simple example we have only included the method for adding an amount of money to the calling object. The outer class `BankAccount` would also have more methods in a real class, but here we have only included methods to deposit an amount of money to the account and to obtain the account balance. Display 13.10 contains a simple demonstration program using the class `BankAccount`.

The class `Money` is a private inner class of the class `BankAccount`. So, the class `Money` cannot be used outside of the class `BankAccount`. (Public inner classes are discussed in Section 13.3 and have some subtleties involved in their use.) Since the class `Money` is local to the class `BankAccount`, the name `Money` can be used for the name of another class outside of the class `BankAccount`. (This would be true even if `Money` were a public inner class.)

Display 13.9 Class with an Inner Class *(Part 1 of 2)*


```
1  public class BankAccount
2  {
3      private class Money
4      {
5          private long dollars;
6          private int cents;

7          public Money(String stringAmount)
8          {
9              abortOnNull(stringAmount);
10             int length = stringAmount.length();
11             dollars = Long.parseLong(
12                         stringAmount.substring(0, length - 3));
13             cents = Integer.parseInt(
14                         stringAmount.substring(length - 2, length));
15         }

16         public String getAmount()
17         {
18             if (cents > 9)
19                return (dollars + "." + cents);
20             else
21                return (dollars + ".0" + cents);
22         }

23         public void addIn(Money secondAmount)
24         {
25             abortOnNull(secondAmount);
26             int newCents = (cents + secondAmount.cents)%100;
27             long carry = (cents + secondAmount.cents)/100;
28             cents = newCents;
29             dollars = dollars + secondAmount.dollars + carry;
30         }

31        private void abortOnNull(Object o)
32        {
33            if (o == null)
34            {
35                 System.out.println("Unexpected null argument.");
36                 System.exit(0);
37            }
38        }
39 }
```

The modifier `private` in this line should not be changed to `public`. However, the modifiers `public` and `private` inside the inner class `Money` can be changed to anything else and it would have no effect on the class `BankAccount`.

The definition of the inner class ends here, but the definition of the outer class continues in Part 2 of this display.

Display 13.9 Class with an Inner Class *(Part 2 of 2)*

```
40      private Money balance;

41      public BankAccount()
42      {
43          balance = new Money("0.00");
44      }

45      public String getBalance()
46      {
47          return balance.getAmount();
48      }

49      public void makeDeposit(String depositAmount)
50      {
51          balance.addIn(new Money(depositAmount));
52      }

53      public void closeAccount()
54      {
55          balance.dollars = 0;
56          balance.cents = 0;
57      }
58  }
```

To invoke a nonstatic method of the inner class outside of the inner class, you need to create an object of the inner class.

This invocation of the inner class method `getAmount()` would be allowed even if the method `getAmount()` were marked as `private`.

Notice that the outer class has access to the private instance variables of the inner class.

This class would normally have more methods, but we have only included the methods we need to illustrate the points covered here.

Display 13.10 Demonstration Program for the Class BankAccount *(Part 1 of 2)*


```
1   public class BankAccountDemo
2   {
3       public static void main(String[] args)
4       {
5           System.out.println("Creating a new account.");
6           BankAccount account = new BankAccount();
7           System.out.println("Account balance now = $"
8                                           + account.getBalance());

9           System.out.println("Depositing $100.00");
10          account.makeDeposit("100.00");
11          System.out.println("Account balance now = $"
12                                          + account.getBalance());
```

Display 13.10 Demonstration Program for the Class BankAccount *(Part 2 of 2)*

```
13          System.out.println("Depositing $99.99");
14          account.makeDeposit("99.99");
15          System.out.println("Account balance now = $"
16                                          + account.getBalance());

17          System.out.println("Depositing $0.01");
18          account.makeDeposit("0.01");
19          System.out.println("Account balance now = $"
20                                          + account.getBalance());

21          System.out.println("Closing account.");
22          account.closeAccount();
23          System.out.println("Account balance now = $"
24                                          + account.getBalance());
25      }
26  }
```

SAMPLE DIALOGUE

```
Creating a new account.
Account balance now = $0.00
Depositing $100.00
Account balance now = $100.00
Depositing $99.99
Account balance now = $199.99
Depositing $0.01
Account balance now = $200.00
Closing account.
Account balance now = $0.00
```

We have made the instance variables in the class Money `public` or `private` following our usual conventions for class members. When we discuss public inner classes, this will be important. However, for use within the outer class (and a private inner class cannot be used anyplace else), there is no difference between `public` and `private` or other member modifiers. All instance variables and all methods of the inner class are public to the outer class no matter whether they are marked `public` or `private` or are left unmarked. Notice the method `closeAccount` of the outer class. It uses the private instance variables `dollars` and `cents` of the inner class.

This is still very much a toy example, but we will have occasion to make serious use of private inner classes when we discuss linked lists in Chapter 14 and when we study Swing GUIs in Chapters 16 and 18.

HELPING INNER CLASSES

You may define a class within another class. The inside class is called an **inner class**. A common and simple use of an inner class is to use it as a helping class for the outer class, in which case the inner class should be marked `private`.

Self-Test Exercises

19. Would the following invocation of `getAmount` in the method `getBalance` of the outer class `BankAccount` still be legal if we change the method `getAmount` of the inner class `Money` from `public` to `private`?

```
public String getBalance()
{
    return balance.getAmount();
}
```

20. Since it does not matter if we make the members of a private inner class public or private, can we simply omit the `public` or `private` modifiers from the instance variables and methods of a private inner class?

21. Would it be legal to add the following method to the inner class `Money` in Display 13.9? Remember, the question is would it be legal, not would it be sensible.

```
public void doubleBalance()
{
    balance.addIn(balance);
}
```

22. Would it be legal to add the following method to the inner class `Money` in Display 13.9? Remember, the question is would it be legal, not would it be sensible.

```
public void doubleBalance2()
{
    makeDeposit(balance.getAmount());
}
```

THE `.class` FILE FOR AN INNER CLASS

When you compile any class in Java, that produces a `.class` file. When you compile a class with an inner class, that compiles both the outer class and the inner class and produces two `.class` files. For example, when you compile the class `BankAccount` in Display 13.9, that produces the following two `.class` files:

`BankAccount.class` and `BankAccount$Money.class`

If `BankAccount` had two inner classes, then three `.class` files would be produced.

Pitfall

Other Uses of Inner Classes

In this section we have shown you how to use an inner class in only one way, namely to create and use objects of the inner class from within the outer class method definitions. There are other ways to use inner classes, but they can involve subtleties. If you intend to use inner classes in any of these other ways, you should consult Section 13.3.

13.3 More about Inner Classes ✣

Something deeply hidden had to be behind things.

Albert Einstein, Note quoted in *New York Times Magazine* (August 2, 1964)

In this section we cover some of the more subtle details about using inner classes. It might be best to treat this section as a reference section and look up the relevant cases as you need them. None of the material in this section is used in the rest of this book.

STATIC INNER CLASSES

A normal (nonstatic) inner class, which is the kind of inner class we have discussed so far, has a connection between each of its objects and the object of the outer class that created the inner class object. Among other things, this allows an inner class definition to reference an instance variable or invoke a method of the outer class. If you do not need this connection, you can make your inner class **static** by adding the `static` modifier to your inner class definition, as illustrated by the following sample beginning of a class definition:

static

```
public class OuterClass
{
    private static class InnerClass
    {
```

A static inner class can have nonstatic instance variables and methods, but an object of a static inner class has no connection to an object of the outer class.

You may encounter situations where you need an inner class to be static. For example, if you create an object of the inner class within a static method of the outer class, then the inner class must be static. This follows from the fact that a nonstatic inner class object must arise from an outer class object.

Also, if you want your inner class to itself have static members, then the inner class must be static.

Since a static inner class has no connection to an object of the outer class, you cannot reference an instance variable or invoke a nonstatic method of the outer class within the static inner class.

To invoke a static method of a static inner class within the outer class, simply preface the method name with the name of the inner class and a dot. Similarly, to name a static variable of a static inner class within the outer class, simply preface the static variable name with the name of the inner class and a dot.

Static Inner Class

A static inner class is one that is not associated with an object of the outer class. It is indicated by including the modifier `static` in its class heading.

Self-Test Exercises

23. Can you have a static method in a nonstatic inner class?
24. Can you have a nonstatic method in a static inner class?

PUBLIC INNER CLASSES

public inner class

If an inner class is marked with the `public` modifier instead of the `private` modifier, then it can be used in all the ways we discussed so far, but it can also be used outside of the outer class.

The way that you create an object of the inner class outside of the outer class is a bit different for static and nonstatic inner classes. We consider the case of a nonstatic inner class first. When creating an object of a nonstatic inner class, you need to keep in mind that every object of the nonstatic inner class is associated with some object of the outer class. To put it another way, to create an object of the inner class, you must start with an object of the outer class. This has to be true, since an object of the inner class may invoke a method of the outer class or reference an instance variable of the outer class, and you cannot have an instance variable of the outer class unless you have an object of the outer class.

For example, *if you change the class* `Money` *in Display 13.9 from private to public*, so the class definition begins

```
public class BankAccount
{
    public class Money
```

then you can use an object of the nonstatic inner class `Money` outside of the class `BankAccount` as illustrated by the following:

```
BankAccount account = new BankAccount();
BankAccount.Money amount =
              account.new Money("41.99");
System.out.println(amount.getAmount());
```

This code produces the output

```
41.99
```

Note that the object `amount` of the inner class `Money` is created starting with an object, `account`, of the outer class `BankAccount`, as follows:

```
BankAccount.Money amount =
              account.new Money("41.99");
```

Also, note that the syntax of the second line is *not*

```
new account.Money("41.99"); //Incorrect syntax
```

Within the definition of the inner class `Money`, an object of the inner class can invoke a method of the outer class. However, this is not true outside of the inner class. Outside of the inner class an object of the inner class can only invoke methods of the inner class. So, we could *not* have continued the previous sample code (which is outside the class `BankAccount` and so outside the inner class `Money`) with the following:

```
System.out.println(amount.getBalance()); //Illegal
```

The meaning of `amount.getBalance()` is clear, but it is still not allowed. If you want something equivalent to `amount.getBalance()`, you should use the corresponding object of the class `BankAccount`; in this case, you would use `account.getBalance()`. (Recall that `account` is the `BankAccount` object used to create the inner class object `amount`.)

Now let's consider the case of a static inner class. You can create objects of a public *static* inner class and do so outside of the inner class, in fact even outside of the outer class. To do so outside of the outer class, the situation is similar to, but not exactly the same as, what we outlined for nonstatic inner classes. Consider the following outline:

```
public class OuterClass
{
    public static class InnerClass
    {
        public void nonstaticMethod()
        { ... }

        public static void staticMethod()
        { ... }
```

```
        Other_Members_of_InnerClass
    }

    Other_Members_of_OuterClass
}
```

You can create an object of the inner class outside of the outer class as in the following example:

```
OuterClass.InnerClass innerObject =
              new OuterClass.InnerClass();
```

Note that the syntax is *not*

```
OuterClass.new InnerClass();
```

This may seem like an apparent inconsistency with the syntax for creating the object of a nonstatic inner class. It may help to keep in mind that for a static inner class, `OuterClass.InnerClass` is a well-specified class name and all the information for the object is in that class name. To remember the syntax for a nonstatic inner class, remember that for that case, the object of the outer class modifies how the `new` operator works to create an object of the inner class.

Once you have created an object of the inner class, the object can invoke a nonstatic method in the usual way. For example:

```
innerObject.nonstaticMethod();
```

You can also use the object of the inner class to invoke a static method in the same way. For example:

```
innerObject.staticMethod();
```

However, it is more common, and clearer, to use class names when invoking a static method. For example:

```
OuterClass.InnerClass.staticMethod();
```

Tip

Referring to a Method of the Outer Class

As we've already noted, if a method is invoked in an inner class and the inner class has no such method, then it is assumed to be an invocation of the method by that name in the outer class. For example, we could add a method `showBalance` to the inner class `Money` in Display 13.9, as outlined in what follows:

```
public class BankAccount
{
```

```
    private class Money
    {
        private long dollars;
        private int cents;

        public void showBalance()
        {
            System.out.println(getBalance());
        }
             ...
    }//End of Money

    public String getBalance()
    {...}
        ...
}//End of BankAccount
```

This invocation of `getBalance` is within the definition of the inner class `Money`. But, the inner class `Money` has no method named `getBalance`, so it is presumed to be the method `getBalance` of the outer class `BankAccount`.

But, suppose the inner class did have a method named `getBalance`; then, this invocation of `getBalance` would be an invocation of the method `getBalance` defined in the inner class.

If both the inner and outer classes have a method named `getBalance`, then you can specify that you mean the method of the outer class as follows:

```
public void showBalance()
{
    System.out.println(
                  BankAccount.this.getBalance());
}
```

The syntax

Outer_Class_Name`.this.`*Method_Name*

always refers to a method of the outer class. In the example, `BankAccount.this` means the `this` of `BankAccount`, as opposed to the `this` of the inner class `Money`.

Self-Test Exercises

25. Consider the following class definition:

```
public class OuterClass
{
    public static class InnerClass
```

```
    {
        public static void someMethod()
        {
            System.out.println("From inside.");
        }
    }

    Other_Members_of_OuterClass
}
```

Write an invocation of the static method `someMethod` that you could use in some class you define.

26. Consider the following class definition:

```
public class Outs
{
    private int outerInt = 100;

    public class Ins
    {
        private int innerInt = 25;

        public void specialMethod()
        {
           System.out.println(outerInt);
           System.out.println(innerInt);
        }
    }

    Other_Members_of_OuterClass
}
```

Write an invocation of the method `specialMethod` with an object of the class `Ins`. Part of this exercise is to create the object of the class `Ins`. This should be code that you could use in some class you define.

NESTING INNER CLASSES

It is legal to nest inner classes within inner classes. The rules are the same as what we have already discussed except that names can get longer. For example, if `A` has a public inner class `B`, and `B` has a public inner class `C`, then the following is valid code:

```
A aObject = new A();
A.B bObject =
             aObject.new B();
A.B.C cObject =
             bObject.new C();
```

INNER CLASSES AND INHERITANCE

Suppose `OuterClass` has an inner class named `InnerClass`. If you derive `DerivedClass` from `OuterClass`, then `DerivedClass` automatically has `InnerClass` as an inner class just as if it were defined within `DerivedClass`.

Just as with any other kind of class in Java, you can make an inner class a derived class of some other class. You can also make the outer class a derived class of a different (or the same) base class.

It is not possible to override the definition of an inner class when you define a derived class of the outer class.

It is also possible to use an inner class as a base class to derive classes, but we will not go into those details in this book; there are some subtleties to worry about.

ANONYMOUS CLASSES

anonymous class

If you wish to create an object but have no need to name the object's class, then you can embed the class definition inside the expression with the `new` operator. These sorts of class definitions are called **anonymous classes** because they have no class name. An expression with an anonymous class definition is, like everything in Java, inside of some class definition. Thus, an anonymous class is an inner class. Before we go into the details of the syntax for anonymous classes, let's say a little about where one might use them.

The most straightforward way to create an object is the following:

```
YourClass anObject = new YourClass();
```

If `new YourClass()` is replaced by some expression that defines the class but does not give the class any name, then there is no name `YourClass` to use to declare the variable `anObject`. So, it does not make sense to use an anonymous class in this situation. However, it can make sense in the following situation:

```
SomeOtherType anObject = new YourClass();
```

Here `SomeOtherType` must be a type such that an object of the class `YourClass` is also an object of `SomeOtherType`. In this situation you can replace `new YourClass()` with an expression including an anonymous class instead of `YourClass`. The type `SomeOtherType` is usually a Java interface.

Here's an example of an anonymous class. Suppose you define the following interface:

```
public interface NumberCarrier
{
    public void setNumber(int value);
    public int getNumber();
}
```

Then the following creates an object using an anonymous class definition:

```
NumberCarrier anObject = new NumberCarrier()
               {
                   private int number;
                   public void setNumber(int value)
                   {
                       number = value;
                   }
                   public int getNumber()
                   {
                       return number;
                   }
                };
```

The part in the braces is the same as the part inside the main braces of a class definition. The closing brace is followed by a semicolon, unlike a class definition. (This is because the entire expression will be used as a Java statement.) The beginning part, repeated below, may seem strange:

```
new NumberCarrier()
```

The `new` is sensible enough but what's the point of `NumberCarrier()`? It looks like this is an invocation of a constructor for `NumberCarrier`. But, `NumberCarrier` is an interface and has no constructors. The meaning of `new NumberCarrier()` is simply

```
implements NumberCarrier
```

So what is being said is that the anonymous class implements the `NumberCarrier` interface and is defined as shown between the main braces.

Display 13.11 shows a simple demonstration with two anonymous class definitions. For completeness we have also repeated the definition of the `NumberCarrier` interface in that display.

Tip

Why Use Inner Classes?

Most simple situations do not need inner classes. However, there are situations for which inner classes are a good solution. For example, suppose you want to have a class with two base classes. That is not allowed in Java. However, you can have an outer class derived from one base class with an inner class derived from the other base class. Since the inner and outer classes have access to each other's instance variables and methods, this can often serve as if it were a class with two base classes.

As another example, if you only need one object of a class and the class definition is very short, many programmers like to use an anonymous class (but I must admit I am not one of them).

Display 13.11 Anonymous Classes *(Part 1 of 2)*

This is just a toy example to demonstrate the Java syntax for anonymous classes.

```
1  public class AnonymousClassDemo
2  {
3      public static void main(String[] args)
4      {
5          NumberCarrier anObject =
6                    new NumberCarrier()
7                    {
8                        private int number;
9                        public void setNumber(int value)
10                       {
11                           number = value;
12                       }
13                       public int getNumber()
14                       {
15                          return number;
16                       }
17                    };
18         NumberCarrier anotherObject =
19                   new NumberCarrier()
20                   {
21                       private int number;
22                       public void setNumber(int value)
23                       {
24                           number = 2*value;
25                       }
26                       public int getNumber()
27                       {
28                           return number;
29                       }
30                   };
31         anObject.setNumber(42);
32         anotherObject.setNumber(42);
33         showNumber(anObject);
34         showNumber(anotherObject);
35         System.out.println("End of program.");
36     }
37     public static void showNumber(NumberCarrier o)
38     {
39         System.out.println(o.getNumber());
40     }
41 }
```

This is still the file AnonymousClassDemo.java.

Display 13.11 Anonymous Classes *(Part 2 of 2)*

SAMPLE DIALOGUE

```
42
84
End of program.
```

This is the file NumberCarrier.java.

```
1  public interface NumberCarrier
2  {
3      public void setNumber(int value);
4      public int getNumber();
5  }
```

When we study *linked lists* in Chapter 14, you will see cases where using an inner class as a helping class makes the linked list class self-contained in a very natural way. We will also use inner classes when defining Graphical User Interfaces (GUIs) in Chapters 16 and 18. But, until you learn what linked lists and GUIs are, these are not likely to be compelling examples.

Self-Test Exercises

27. Suppose we replace

    `NumberCarrier anObject`

    with

    `Object anObject`

    in Display 13.11. What would be the first statement in the program to cause an error message? Would it be a compiler error message or a run-time error message?

Chapter Summary

- An interface is a property of a class that says what methods a class that implements the interface must have.
- An interface is defined the same way as a class is defined except that the keyword `interface` is used in place of the keyword `class` and method bodies are replaced by semicolons.

- An interface may not have any instance variables, with one exception: An interface may have defined constants. If you use the syntax for an instance variable in an inner class, the variable is automatically a constant, not a real instance variable.
- An inner class is a class defined within another class.
- One simple use of an inner class is as a helping class to be used in the definition of the outer class methods and/or instance variables.
- A static inner class is one that is not associated with an object of the outer class. It must include the modifier `static` in its class heading.
- To create an object of a nonstatic inner class outside the definition of the outer class, you must first create an object of the outer class and use it to create an object of the inner class.

Answers to Self-Test Exercises

1. Yes to both. An interface is a type and can be used like any other type.
2. Yes. Any of the interface methods that it does not fully define must be made abstract methods.
3. A derived class can have only one base class, but it can implement any number of interfaces.
4. No, but the way to accomplish the same thing is to have one interface extend the other.

   These exercises are for the material on the `Comparable` interface.

5. Yes, the dialog would be the same. The change from the parameter type `Comparable[]` to `Object[]` in the method `interchange` is in fact a good idea,
6. No. This will compile without any error messages. However, the less-than ordering does not satisfy the semantics of the `Comparable` interface. For example, the trichotomy law does not hold.
7. Yes. The three required conditions are true for objects of the class `Circle`:

   (Irreflexive) By definition, no circle is inside itself.

   (Trichotomy) For any two circles `c1` and `c2` with centers at the origin, one and only one of the following holds true: `c1` is inside of `c2`, `c2` is inside of `c1`, or `c1` equals `c2`.

   (Transitivity) If `c1` is inside of `c2` and `c2` is inside of `c3`, then `c1` is inside of `c3`.
8. The class will produce a compiler error message saying that there is an inconsistency in the definitions of `ANSWER`.
9. The class will compile and run with no error messages. Since the named constant `ANSWER` is never used, there is no inconsistency.
10. The class will produce a compiler error message saying that you have not implemented the heading for `getStuff` in `InterfaceA`.
11. They will all compile and the program will run. The two definitions of `getStuff` have different numbers of parameters and so this is overloading. There is no inconsistency.

12.
```
public class StockItem implements Cloneable
{
    private int number;
    private String name;
    public void setNumber(int newNumber)
    {
        number = newNumber;
    }
       ...
    public Object clone()
    {
       try
       {
          return super.clone();
       }
       catch(CloneNotSupportedException e)
       {//This should not happen.
          return null; //To keep compiler happy.
       }
    }
}
```

13. Note that you do not catch a `CloneNotSupportedException` because any such thrown exception in `super.clone` is caught inside the base class method `super.clone`.

```
public class PricedItem extends StockItem
                        implements Cloneable
{
    private double price;

       ...
    public Object clone()
    {
       return super.clone();
    }
}
```

14.
```
public class Record implements Cloneable
{
    private StockItem item1;
    private StockItem item2;
    private String description;

       ...
    public Object clone()
    {
        try
        {
            Record copy =
```

```
                    (Record)super.clone();
            copy.item1 =
                   (StockItem)item1.clone();
            copy.item2 =
                   (StockItem)item2.clone();
            return copy;
        }
        catch(CloneNotSupportedException e)
        {//This should not happen.
            return null; //To keep compiler happy.
        }
    }
}
```

15. Note that you do not catch a `CloneNotSupportedException` because any such thrown exception in `super.clone` is caught inside the base class method `super.clone`.

```
public class BigRecord extends Record
                        implements Cloneable
{
    private StockItem item3;

       ...
    public Object clone()
    {
        BigRecord copy =
                  (BigRecord)super.clone();
        copy.item3 =
                   (StockItem)item3.clone();
        return copy;
    }
}
```

16. The heading of the class definition changes to what is shown in the following and the method `clone` shown there is added. The version of `Date` for this chapter on the accompanying CD includes this definition of `clone`.

extra code on CD

```
public class Date implements Cloneable
{
    private String month;
    private int day;
    private int year;

       ...
    public Object clone()
    {
       try
       {
          return super.clone();//Invocation of
```

```
                //clone in the base class Object
        }
        catch(CloneNotSupportedException e)
        {//This should not happen.
           return null; //To keep compiler happy.
        }
    }
}
```

17. The heading of the class definition changes to what is shown in the following and the method `clone` shown there is added. The version of `Date` for this chapter on the accompanying CD includes this definition of `clone`.

extra code on CD

```
public class Employee implements Cloneable
{
    private String name;
    private Date hireDate;


       ...
    public Object clone()
    {
        try
        {
            Employee copy =
                      (Employee)super.clone();
            copy.hireDate =
                   (Date)hireDate.clone();
            return copy;
        }
        catch(CloneNotSupportedException e)
        {//This should not happen.
            return null; //To keep compiler happy.
        }
    }
}
```

18. The heading of the class definition changes to what is shown in the following and the method `clone` shown there is added. Note that you do not catch a `CloneNotSupported-Exception` because any such thrown exception in `super.clone` is caught inside the base class method `super.clone`. The version of `HourlyEmployee` for this chapter on the accompanying CD includes this definition of `clone`.

extra code on CD

```
public class HourlyEmployee extends Employee
                            implements Cloneable
{
    private double wageRate ;
    private double hours;
```

```
    ...
  public Object clone()
  {
      HourlyEmployee copy =
                (HourlyEmployee)super.clone();
      return copy;
  }
}
```

19. It would still be legal. An outer class has access to all the private members of an inner class.
20. Yes, they can be omitted, but the reason is that it indicates package access, and in a private inner class, all privacy modifiers, including package access, are equivalent to public. (Note that the situation for public inner classes will be different.)
21. Yes, it is legal to add the method `doubleBalance` to the inner class `Money` because an inner class has access to the instance variables, like `balance` of the outer class. To test this out, add the following as a method of the outer class:

    ```
    public void test()
    {
        balance.doubleBalance();
    }
    ```

22. It would be legal. The method `makeDeposit` is assumed to be the method `makeDeposit` of the outer class. The calling object `balance` is assumed to be the instance variable of the outer class. These sorts of tricks can lead to confusing code. So, use them sparingly. This is just an exercise.
23. No, a nonstatic inner class cannot have any static methods.
24. Yes, you can have a nonstatic method in a static inner class.
25. `OuterClass.InnerClass.someMethod();`
26. ```
    Outs outerObject = new Outs();
    Outs.Ins innerObject =
                  outerObject.new Ins();
    innerObject.specialMethod();
    ```
27. You would get your first error on the following statement and it would be a complier error:

    ```
    anObject.setNumber(42);
    ```

    With the change described in the exercise, `anObject` is of type `Object` and `Object` has no method named `setNumber`.
    ```

PROGRAMMING PROJECTS

1. In Display 13.5 we gave a sorting method to sort an array of type `Comparable[]`. In Display 12.6 we gave a sorting method that used the merge sort algorithm to sort an array of type `double[]` into increasing order. Redo the method in Display 12.6 so it applies to an array of type `Comparable[]`. Also, do a suitable test program.

2. In Display 13.5 we gave a sorting method to sort an array of type `Comparable[]`. In Display 12.8 we gave a sorting method that used the quick sort algorithm to sort an array of type `double[]` into increasing order. Redo the method in Display 12.8 so it applies to an array of type `Comparable[]`. Also, do a suitable test program.

3. Redo the class `Person` in Display 5.11 so that it implements the `Cloneable` interface. This will require that you also redo the class `Date` so it implements the `Cloneable` interface. Also, do a suitable test program.

4. Redo the class `Person` in Display 5.11 so that the class `Date` is a private inner class of the class `Person`. Also, do a suitable test program. (You need not start from the version produced in Programming Project 3. You can ignore Programming Project 3 when you do this project.)

5. This is a combination of Programming Projects 3 and 4. Redo the class `Person` in Display 5.11 so that the class `Date` is a private inner class of the class `Person` and so that the class `Person` implements the `Cloneable` interface. Also, do a suitable test program.

6. Redo the class `Employee` and the class `HourlyEmployee` in Displays 7.2 and 7.3 so that the class `Date` is an inner class of the class `Employee` and an inherited inner class of the class `HourlyEmployee`. Also, do a suitable test program.