Chapter 6

Interfaces and Inner Classes

You have now seen all the basic tools for object-oriented programming in Java. This chapter shows you two advanced techniques that are very commonly used. Despite their less obvious nature, you will need to master them to complete your Java tool chest.

The first, called an *interface*, is a way of describing *what* classes should do, without specifying *how* they should do it. A class can *implement* one or more interfaces. You can then use objects of these implementing classes anytime that conformance to the interface is required. After we cover interfaces, we take up cloning an object (or deep copying, as it is sometimes called). A clone of an object is a new object that has the same state as the original but a different identity. In particular, you can modify the clone without affecting the original. Finally, we move on to the mechanism of *inner classes*. Inner classes are technically somewhat complex—they are defined inside other classes, and their methods can access the fields of the surrounding class. Inner classes are useful when you design collections of cooperating classes. In particular, inner classes are important to write concise, professional-looking code to handle graphical user interface events.

This chapter concludes with a discussion of *proxies*, objects that implement arbitrary interfaces. A proxy is a very specialized construct that is useful for building system-level tools. You can safely skip that section on first reading.

Interfaces

In the Java programming language, an interface is not a class but a set of *requirements* for classes that want to conform to the interface.

Typically, the supplier of some service states: "If your class conforms to a particular interface, then I'll perform the service." Let's look at a concrete example. The `sort` method of the `Arrays` class promises to sort an array of objects, but under one condition: The objects must belong to classes that implement the `Comparable` interface.

Here is what the `Comparable` interface looks like:

```
public interface Comparable
{
   int compareTo(Object other);
}
```

This means that any class that implements the `Comparable` interface is required to have a `compareTo` method, and the method must take an `Object` parameter and return an integer.

All methods of an interface are automatically `public`. For that reason, it is not necessary to supply the keyword `public` when declaring a method in an interface.

Of course, there is an additional requirement that the interface cannot spell out: When calling `x.compareTo(y)`, the `compareTo` method must actually be able to compare two objects and return an indication whether `x` or `y` is larger. The method is supposed to return a negative number if `x` is smaller than `y`, zero if they are equal, and a positive number otherwise.

This particular interface has a single method. Some interfaces have more than one method. As you will see later, interfaces can also define constants. What is more important, however, is what interfaces *cannot* supply. Interfaces never have instance fields, and the methods are never implemented in the interface. Supplying instance fields and method implementations is the job of the classes that implement the interface. You can think of an interface as being similar to an abstract class with no instance fields. However, there are some differences between these two concepts—we will look at them later in some detail.

Now suppose we want to use the `sort` method of the `Arrays` class to sort an array of `Employee` objects. Then the `Employee` class must *implement* the `Comparable` interface.

To make a class implement an interface, you have to carry out two steps:

1. You declare that your class intends to implement the given interface.
2. You supply definitions for all methods in the interface.

To declare that a class implements an interface, use the `implements` keyword:

```
class Employee implements Comparable
```

Of course, now the `Employee` class needs to supply the `compareTo` method. Let's suppose that we want to compare employees by their salary. Here is a `compareTo` method that returns –1 if the first employee's salary is less than the second employee's salary, 0 if they are equal, and 1 otherwise.

```
public int compareTo(Object otherObject)
{
   Employee other = (Employee)otherObject;
   if (salary < other.salary) return -1;
   if (salary > other.salary) return 1;
   return 0;
}
```

In the interface declaration, the `compareTo` method was not declared `public` because all methods in an *interface* are automatically public. However, when implementing the interface, you must declare the method as `public`. Otherwise, the compiler assumes that the method has package visibility—the default for a *class*. Then the compiler complains that you try to supply a weaker access privilege.

The `compareTo` method of the `Comparable` interface returns an integer. If the objects are not equal, it does not matter what negative or positive value you return. This flexibility can be useful when comparing integer fields. For example, suppose each employee has a unique integer `id`, and you want to sort by employee ID number. Then you can simply return `id - other.id`. That value will be some negative value if the first ID number is less than the other, 0 if they are the same ID, and some positive value otherwise. However, there is one caveat: The range of the integers must be small enough that the subtraction does not overflow. If you know that the IDs are not negative or that their absolute value is at most `(Integer.MAX_VALUE - 1) / 2`, you are safe.

Of course, the subtraction trick doesn't work for floating-point numbers. The difference `salary - other.salary` can round to 0 if the salaries are close together but not identical.

Now you saw what a class must do to avail itself of the sorting service—it must implement a `compareTo` method. That's eminently reasonable. There needs to be some way for the `sort` method to compare objects. But why can't the `Employee` class simply provide a `compareTo` method without implementing the `Comparable` interface?

The reason for interfaces is that the Java language is *strongly typed*. When making a method call, the compiler needs to be able to check that the method actually exists. Somewhere in the `sort` method, there will be statements like this:

```
if (a[i].compareTo(a[j]) > 0)
{
   // rearrange a[i] and a[j]
   . . .
}
```

The compiler must know that `a[i]` actually has a `compareTo` method. If `a` is an array of `Comparable` objects, then the existence of the method is assured, because every class that implements the `Comparable` interface must supply the method.

You would expect that the `sort` method in the `Arrays` class is defined to accept a `Comparable[]` array, so that the compiler can complain if anyone ever calls `sort` with an array whose element type doesn't implement the `Comparable` interface. Sadly, that is not the case. Instead, the `sort` method accepts an `Object[]` array and uses a clumsy cast:

```
// from the standard library--not recommended
if (((Comparable)a[i]).compareTo((Comparable)a[j]) > 0)
{
   // rearrange a[i] and a[j]
   . . .
}
```

If `a[i]` does not belong to a class that implements the `Comparable` interface, then the virtual machine throws an exception. (Note that the second cast to `Comparable` is not necessary because the explicit parameter of the `compareTo` method has type `Object`, not `Comparable`.)

See Example 6–1 for the full code for sorting of an employee array.

Example 6–1: EmployeeSortTest.java

```
1. import java.util.*;
2.
3. public class EmployeeSortTest
4. {  public static void main(String[] args)
5.    {  Employee[] staff = new Employee[3];
6.
7.       staff[0] = new Employee("Harry Hacker", 35000);
```

```
8.       staff[1] = new Employee("Carl Cracker", 75000);
9.       staff[2] = new Employee("Tony Tester", 38000);
10.
11.      Arrays.sort(staff);
12.
13.      // print out information about all Employee objects
14.      for (int i = 0; i < staff.length; i++)
15.      {  Employee e = staff[i];
16.         System.out.println("name=" + e.getName()
17.            + ",salary=" + e.getSalary());
18.      }
19.   }
20. }
21.
22. class Employee implements Comparable
23. {  public Employee(String n, double s)
24.   {  name = n;
25.      salary = s;
26.   }
27.
28.   public String getName()
29.   {  return name;
30.   }
31.
32.   public double getSalary()
33.   {  return salary;
34.   }
35.
36.   public void raiseSalary(double byPercent)
37.   {  double raise = salary * byPercent / 100;
38.      salary += raise;
39.   }
40.
41.   /**
42.      Compares employees by salary
43.      @param otherObject another Employee object
44.      @return a negative value if this employee has a lower
45.      salary than otherObject, 0 if the salaries are the same,
46.      a positive value otherwise
47.   */
48.   public int compareTo(Object otherObject)
49.   {  Employee other = (Employee)otherObject;
50.      if (salary < other.salary) return -1;
51.      if (salary > other.salary) return 1;
52.      return 0;
53.   }
54.
55.   private String name;
56.   private double salary;
57. }
```

java.lang.Comparable 1.0

- `int compareTo(Object otherObject)`
  compares this object with `otherObject` and returns a negative integer if this object is less than `otherObject`, zero if they are equal, and a positive integer otherwise.

According to the language standard: "The implementor must ensure sgn(`x.compareTo(y)`) = -sgn(`y.compareTo(x)`) for all `x` and `y`. (This implies that `x.compareTo(y)` must throw an exception if `y.compareTo(x)` throws an exception.)" Here, "sgn" is the *sign* of a number: sgn(n) is -1 if n is negative, 0 if n equals 0, and 1 if n is positive. In plain English, if you flip the parameters of `compareTo`, the sign (but not necessarily the actual value) of the result must also flip. That's not a problem, but the implication about exceptions is tricky. Suppose `Manager` has its own comparison method that compares two managers. It might start like this:

```
public int compareTo(Object otherObject)
{
   Manager other = (Manager)otherObject;
   . . .
}
```

That violates the "antisymmetry" rule. If `x` is an `Employee` and `y` is a `Manager`, then the call `x.compareTo(y)` doesn't throw an exception—it simply compares `x` and `y` as employees. But the reverse, `y.compareTo(x)` throws a `ClassCastException`.

The same issue comes up when programming an `equals` method. However, in that case, you simply test if the two classes are identical, and if they aren't, you know that you should return `false`. However, if `x` and `y` aren't of the same class, it is not clear whether `x.compareTo(y)` should return a negative or a positive value. Maybe managers think that they should compare larger than any employee, no matter what the salary. But then they need to explicitly implement that check.

If you don't trust the implementors of your subclasses to grasp this subtlety, you can declare `compareTo` as a `final` method. Then the problem never arises because subclasses can't supply their own version. Conversely, if you implement a `compareTo` method of a subclass, you need to provide a thorough test. Here is an example:

```
if (otherObject instanceof Manager)
{
   Manager other = (Manager)otherObject;
   . . .
}
else if (otherObject instanceof Employee)
{
   return 1; // managers are always better :-(
}
else
   return -((Comparable)otherObject).compareTo(this);
```

java.util.Arrays 1.2

- `static void sort(Object[] a)`
  sorts the elements in the array `a`, using a tuned mergesort algorithm. All elements in the array must belong to classes that implement the `Comparable` interface, and they must all be comparable to each other.

Properties of Interfaces

Interfaces are not classes. In particular, you can never use the `new` operator to instantiate an interface:

```
x = new Comparable(. . .); // ERROR
```

However, even though you can't construct interface objects, you can still declare sinterface variables.

```
Comparable x; // OK
```

An interface variable must refer to an object of a class that implements the interface:

```
x = new Employee(. . .);
   // OK provided Employee implements Comparable
```

Next, just as you use `instanceof` to check if an object is of a specific class, you can use `instanceof` to check if an object implements an interface:

```
if (anObject instanceof Comparable) { . . . }
```

Just as you can build hierarchies of classes, you can extend interfaces. This allows for multiple chains of interfaces that go from a greater degree of generality to a greater degree of specialization. For example, suppose you had an interface called `Moveable`.

```
public interface Moveable
{
   void move(double x, double y);
}
```

Then, you could imagine an interface called `Powered` that extends it:

```
public interface Powered extends Moveable
{
   double milesPerGallon();
}
```

Although you cannot put instance fields or static methods in an interface, you can supply constants in them. For example:

```
public interface Powered extends Moveable
{
   double milesPerGallon();
   double SPEED_LIMIT = 95; // a public static final constant
}
```

Just as methods in an interface are automatically `public`, fields are always `public static final`.

It is legal to tag interface methods as `public`, and fields as `public static final`. Some programmers do that, either out of habit or for greater clarity. However, the Java Language Specification recommends not to supply the redundant keywords, and we follow that recommendation.

Some interfaces define just constants and no methods. For example, the standard library contains an interface `SwingConstants` that defines constants `NORTH`, `SOUTH`, `HORIZONTAL`, and so on. Any class that chooses to implement the `SwingConstants` interface automatically inherits these constants. Its methods can simply refer to `NORTH` rather than the more cumbersome `SwingConstants.NORTH`.

While each class can only have one superclass, classes can implement *multiple* interfaces. This gives you the maximum amount of flexibility in defining a class's behavior. For example, the Java programming language has an important interface built into it, called `Cloneable`. (We will discuss this interface in detail in the next section.) If your class implements `Cloneable`, the `clone` method in the `Object` class will make an exact copy of your class's objects. Suppose, therefore, you want cloneability and comparability. Then you simply implement both interfaces.

```
class Employee implements Cloneable, Comparable
```

Use commas to separate the interfaces that describe the characteristics that you want to supply.

Interfaces and Abstract Classes

If you read the section about abstract classes in Chapter 5, you may wonder why the designers of the Java programming language bothered with introducing the concept of interfaces. Why can't `Comparable` simply be an abstract class:

```
abstract class Comparable // why not?
{
```


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

Then the `Employee` class would simply extend this abstract class and supply the `compareTo` method:

```
class Employee extends Comparable // why not?
{
   public int compareTo(Object other) { . . . }
}
```

There is, unfortunately, a major problem with using an abstract base class to express a generic property. A class can only extend a single class. Suppose that the `Employee` class already extends a different class, say `Person`. Then it can't extend a second class.

```
class Employee extends Person, Comparable // ERROR
```

But each class can implement as many interfaces as it likes:

```
class Employee extends Person implements Comparable // OK
```

Other programming languages, in particular C++, allow a class to have more than one superclass. This feature is called *multiple inheritance.* The designers of Java chose not to support multiple inheritance because it makes the language either very complex (as in C++) or less efficient (as in Eiffel).

Instead, interfaces give most of the benefits of multiple inheritance while avoiding the complexities and inefficiencies.

C++ has multiple inheritance and all the complications that come with it, such as virtual base classes, dominance rules, and transverse pointer casts. Few C++ programmers use multiple inheritance, and some say it should never be used. Other programmers recommend using multiple inheritance only for "mix-in" style inheritance. In the mix-in style, a primary base class describes the parent object, and additional base classes (the so-called mix-ins) may supply auxiliary characteristics. That style is similar to a Java class with a single base class and additional interfaces. However, in C++, mix-ins can add default behavior, whereas Java interfaces cannot.

Microsoft has long been a proponent of using interfaces instead of using multiple inheritance. In fact, the Java notion of an interface is essentially equivalent to how Microsoft's COM technology uses interfaces. As a result of this unlikely convergence of minds, it is easy to supply tools based on the Java programming language to build COM objects (such as ActiveX controls). This is done (pretty much transparently to the coder) in, for example, Microsoft's J++ product and is also the basis for Sun's JavaBeans-to-ActiveX bridge.

Interfaces and Callbacks

A common pattern in programming is the *callback* pattern. In this pattern, you want to specify the action that should occur whenever a particular event happens. For example, you may want a particular action to occur when a button is clicked or a menu item is selected. However, since you have not yet seen how to implement user interfaces, we will consider a similar but simpler situation.

The `javax.swing` class contains a `Timer` class that is useful if you want to be notified whenever a time interval has elapsed. For example, if a part of your program contains a clock, then you can ask to be notified every second so that you can update the clock face.

When you construct a timer, you set the time interval, and you tell it what it should do whenever the time interval has elapsed.

How do you tell the timer what it should do? In many programming languages, you supply the name of a function that the timer should call periodically. However, the classes

in the Java standard library take an object-oriented approach. You pass an object of some class. The timer then calls one of the methods on that object. Passing an object is more flexible than passing a function because the object can carry additional information.

Of course, the timer needs to know what method to call. The timer requires that you specify an object of a class that implements the `ActionListener` interface of the `java.awt.event` package. Here is that interface:

```
public interface ActionListener
{
   void actionPerformed(ActionEvent event);
}
```

The timer calls the `actionPerformed` method when the time interval has expired.

As you saw in Chapter 5, Java does have the equivalent of function pointers, namely, `Method` objects. However, they are difficult to use, slower, and cannot be checked for type safety at compile time. Whenever you would use a function pointer in C++, you should consider using an interface in Java.

Suppose you want to print a message "At the tone, the time is . . .," followed by a beep, once every ten seconds. You need to define a class that implements the `ActionListener` interface. Then place whatever statements you want to have executed inside the `actionPerformed` method.

```
class TimePrinter implements ActionListener
{
   public void actionPerformed(ActionEvent event)
   {
      Date now = new Date();
      System.out.println("At the tone, the time is " + now);
      Toolkit.getDefaultToolkit().beep();
   }
}
```

Note the `ActionEvent` parameter of the `actionPerformed` method. This parameter gives information about the event, such as the source object that generated it—see Chapter 8 for more information. However, detail information about the event is not important in this program, and you can safely ignore the parameter.

Next, you construct an object of this class and pass it to the `Timer` constructor.

```
ActionListener listener = new TimePrinter();
Timer t = new Timer(10000, listener);
```

The first parameter of the `Timer` constructor is the time interval that must elapse between notifications, measured in milliseconds. We want to be notified every ten seconds. The second parameter is the listener object.

Finally, you start the timer.

```
t.start();
```

Every ten seconds, a message like

```
At the tone, the time is Thu Apr 13 23:29:08 PDT 2000
```

is displayed, followed by a beep.

Example 6–2 puts the timer and its action listener to work. After the timer is started, the program puts up a message dialog and waits for the user to click the Ok button to stop. While the program waits for the user, the current time is displayed in ten second intervals.

Be patient when running the program. The "Quit program?" dialog box appears right away, but the first timer message is displayed after ten seconds.

Note that the program imports the `javax.swing.Timer` class by name, in addition to importing `javax.swing.*` and `java.util.*`. This breaks the ambiguity between `javax.swing.Timer` and `java.util.Timer`, an unrelated class for scheduling background tasks.

Example 6–2: TimerTest.java

```
1. import java.awt.*;
2. import java.awt.event.*;
3. import java.util.*;
4. import javax.swing.*;
5. import javax.swing.Timer;
6. // to resolve conflict with java.util.Timer
7.
8. public class TimerTest
9. {
10.    public static void main(String[] args)
11.    {
12.       ActionListener listener = new TimePrinter();
13.
14.       // construct a timer that calls the listener
15.       // once every 10 seconds
16.       Timer t = new Timer(10000, listener);
17.       t.start();
18.
19.       JOptionPane.showMessageDialog(null, "Quit program?");
20.       System.exit(0);
21.    }
22. }
23.
24. class TimePrinter implements ActionListener
25. {
26.    public void actionPerformed(ActionEvent event)
27.    {
28.       Date now = new Date();
29.       System.out.println("At the tone, the time is " + now);
30.       Toolkit.getDefaultToolkit().beep();
31.    }
32. }
```

javax.swing.JOptionPane 1.2

- `static void showMessageDialog(Component parent, Object message)`
  displays a dialog box with a message prompt and an Ok button. The dialog is centered over the `parent` component. If `parent` is `null`, the dialog is centered on the screen.

javax.swing.Timer 1.2

- `Timer(int interval, ActionListener listener)`
  constructs a timer that notifies `listener` whenever `interval` milliseconds have elapsed.
- `void start()`
  starts the timer. Once started, the timer calls `actionPerformed` on its listeners.
- `void stop()`
  stops the timer. Once stopped, the timer no longer calls `actionPerformed` on its listeners

javax.awt.Toolkit 1.0

- `static Toolkit getDefaultToolkit()`
  gets the default toolkit. A toolkit contains information about the graphical user interface environment.
- `void beep()`
  Emits a beep sound.

Object Cloning

When you make a copy of a variable, the original and the copy are references to the same object. (See Figure 6–1.) This means a change to either variable also affects the other.

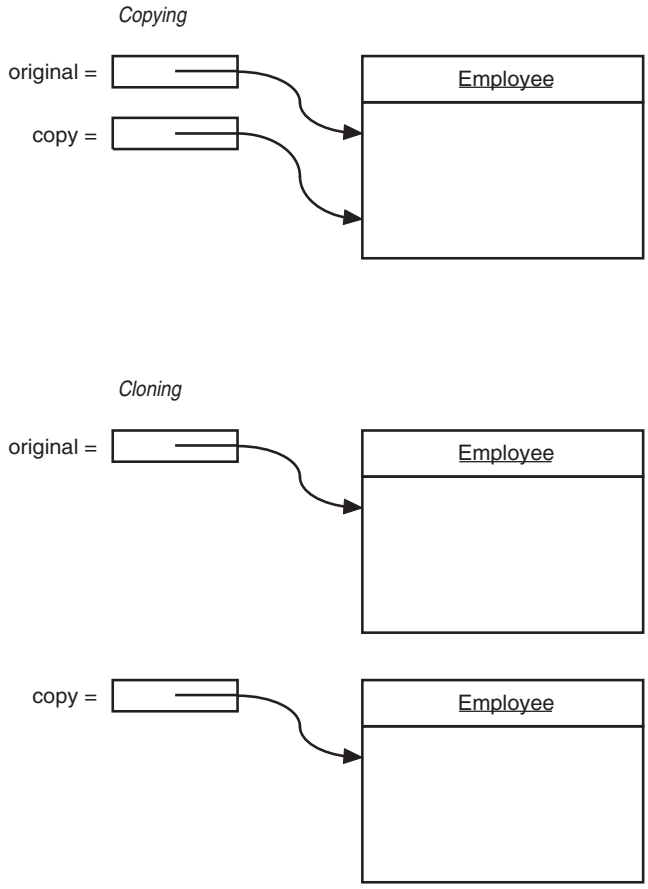

Figure 6–1: Copying and cloning

```
Employee original = new Employee("John Public", 50000);
Employee copy = original;
copy.raiseSalary(10); // oops--also changed original
```

If you would like `copy` to be a new object that begins its life being identical to `original` but whose state can diverge over time, then you use the `clone()` method.

```
Employee copy = (Employee)original.clone();
   // must cast—clone returns an Object
copy.raiseSalary(10); // OK—original unchanged
```

But it isn't quite so simple. The `clone` method is a `protected` method of `Object`, which means that your code cannot simply call it. Only the `Employee` class can clone `Employee` objects. There is a reason for this restriction. Think about the way in which the `Object` class can implement `clone`. It knows nothing about the object at all, so it can make only a field-by-field copy. If all data fields in the object are numbers or other basic types, copying the fields is just fine. But if the object contains references to subobjects, then copying the field gives you another reference to the subobject, so the original and the cloned objects still share some information.

To visualize that phenomenon, let's consider the `Employee` class that was introduced in Chapter 4. Figure 6–2 shows what happens when you use the `clone` method of the `Object` class to clone such an `Employee` object. As you can see, the default cloning operation is "shallow"—it doesn't clone objects that are referenced inside other objects.

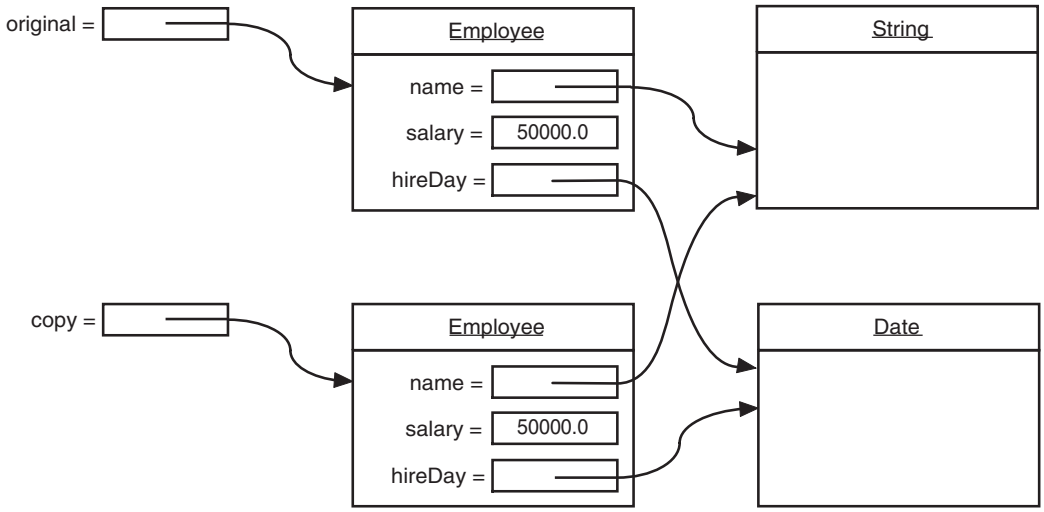

Figure 6–2: A shallow copy

Does it matter if the copy is shallow? It depends. If the subobject that is shared between the original and the shallow clone is *immutable*, then the sharing is safe. This happens in two situations. The subobject may belong to an immutable class, such as `String`. Alternatively, the subobject may simply remain constant throughout the lifetime of the object, with no mutators touching it and no methods yielding a reference to it.

Quite frequently, however, subobjects are mutable, and you must redefine the `clone` method to make a *deep copy* that clones the subobjects as well. In our example, the `hireDay` field is a `Date`, which is mutable.

For every class, you need to decide whether or not

1. The default `clone` method is good enough;

2. The default `clone` method can be patched up by calling `clone` on the mutable subobjects;
3. `clone` should not be attempted.

The third option is actually the default. To choose either the first or the second option, a class must

1. Implement the `Cloneable` interface, and
2. Redefine the `clone` method with the `public` access modifier.

The `clone` method is declared `protected` in the `Object` class so that your code can't simply call `anObject.clone()`. But aren't protected methods accessible from any subclass, and isn't every class a subclass of `Object`? Fortunately, the rules for protected access are more subtle (see Chapter 5). A subclass can call a protected `clone` method only to clone *its own* objects. You must redefine `clone` to be public to allow objects to be cloned by any method.

In this case, the appearance of the `Cloneable` interface has nothing to do with the normal use of interfaces. In particular, it does *not* specify the `clone` method—that method is inherited from the `Object` class. The interface merely serves as a tag, indicating that the class designer understands the cloning process. Objects are so paranoid about cloning that they generate a checked exception if an object requests cloning but does not implement that interface.

The `Cloneable` interface is one of a handful of *tagging interfaces* that Java provides. Recall that the usual purpose of an interface such as `Comparable` is to ensure that a class implements a particular method or set of methods. A tagging interface has no methods; its only purpose is to allow the use of `instanceof` in a type inquiry:

```
if (obj instanceof Cloneable) . . .
```

Even if the default (shallow copy) implementation of `clone` is adequate, you still need to implement the `Cloneable` interface, redefine `clone` to be public, call `super.clone()` and catch the `CloneNotSupportedException`. Here is an example:

```
class Employee implements Cloneable
{
   public Object clone() // raise visibility level to public
   {
      try
      {
         return super.clone();
      }
      catch (CloneNotSupportedException e) { return null; }
      // this won't happen, since we are Cloneable
   }

   . . .
}
```

The `clone` method of the `Object` class threatens to throw a `CloneNotSupportedException`—it does that whenever `clone` is invoked on an object whose class does not implement the `Cloneable` interface. Of course, the `Employee` class implements the `Cloneable` interface, so the exception won't be thrown. However, the compiler does not know that. Therefore, you still need to catch the exception and return a dummy value.

The `clone` method that you just saw adds no functionality to the shallow copy provided by `Object.clone`. It merely makes the method public. To make a deep copy, you have to work harder and clone the mutable instance fields.

Here is an example of a `clone` method that creates a deep copy:

```
class Employee implements Cloneable
{
   . . .
   public Object clone()
   {
      try
      {
         // call Object.clone()
         Employee cloned = (Employee)super.clone();

          // clone mutable fields
         cloned.hireDay = (Date)hireDay.clone()

         return cloned;
      }
      catch (CloneNotSupportedException e) { return null; }
   }
}
```

Users of your `clone` method still have to cast the result. The `clone` method always has return type `Object`.

As you can see, cloning is a subtle business, and it makes sense that it is defined as `protected` in the `Object` class. (See Chapter 12 for an elegant method for cloning objects, using the object serialization feature of Java.)

When you define a `public clone` method, you have lost a safety mechanism. Your `clone` method is inherited by the subclasses, whether or not it makes sense for them. For example, once you have defined the `clone` method for the `Employee` class, anyone can also clone `Manager` objects. Can the `Employee` clone method do the job? It depends on the fields of the `Manager` class. In our case, there is no problem because the `bonus` field has primitive type. But in general, you need to make sure to check the `clone` method of any class that you extend.

The program in Example 6–3 clones an `Employee` object, then invokes two mutators. The `raiseSalary` method changes the value of the `salary` field, while the `setHireDay` method changes the state of the `hireDay` field. Neither mutation affects the original object because `clone` has been defined to make a deep copy.

Example 6–3: CloneTest.java

```
1. import java.util.*;
2.
3. public class CloneTest
4. {
5.     public static void main(String[] args)
6.     {
7.        Employee original = new Employee("John Q. Public", 50000);
8.        original.setHireDay(2000, 1, 1);
9.        Employee copy = (Employee)original.clone();
10.       copy.raiseSalary(10);
11.       copy.setHireDay(2002, 12, 31);
12.       System.out.println("original=" + original);
13.       System.out.println("copy=" + copy);
14.    }
```

```
15. }
16.
17. class Employee implements Cloneable
18. {
19.     public Employee(String n, double s)
20.     {
21.        name = n;
22.        salary = s;
23.     }
24.
25.     public Object clone()
26.     {
27.        try
28.        {
29.           // call Object.clone()
30.           Employee cloned = (Employee)super.clone();
31.
32.           // clone mutable fields
33.           cloned.hireDay = (Date)hireDay.clone();
34.
35.           return cloned;
36.        }
37.        catch (CloneNotSupportedException e) { return null; }
38.     }
39.
40.     /**
41.        Set the pay day to a given date
42.        @param year the year of the pay day
43.        @param month the month of the pay day
44.        @param day the day of the pay day
45.     */
46.     public void setHireDay(int year, int month, int day)
47.     {
48.        hireDay = new GregorianCalendar(year,
49.           month - 1, day).getTime();
50.     }
51.
52.     public void raiseSalary(double byPercent)
53.     {
54.        double raise = salary * byPercent / 100;
55.        salary += raise;
56.     }
57.
58.     public String toString()
59.     {
60.        return "Employee[name=" + name
61.           + ",salary=" + salary
62.           + ",hireDay=" + hireDay
63.           + "]";
64.     }
65.
66.     private String name;
67.     private double salary;
68.     private Date hireDay;
69. }
```

Inner Classes

An *inner class* is a class that is defined inside another class. Why would you want to do that? There are four reasons:

- An object of an inner class can access the implementation of the object that created it—including data that would otherwise be private.
- Inner classes can be hidden from other classes in the same package.
- *Anonymous* inner classes are handy when you want to define callbacks on the fly.
- Inner classes are very convenient when you are writing event-driven programs.

You will soon see examples that demonstrate the first three benefits. (For more information on the event model, please turn to Chapter 8.)

C++ has *nested classes*. A nested class is contained inside the scope of the enclosing class. Here is a typical example: a linked list class defines a class to hold the links, and a class to define an iterator position.

```
class LinkedList
{
public:
   class Iterator // a nested class
   {
   public:
      void insert(int x);
      int erase();
      . . .
   };
   . . .
private:
   class Link // a nested class
   {
   public:
      Link* next;
      int data;
   };
   . . .
};
```

The nesting is a relationship between *classes*, not *objects*. A `LinkedList` object does *not* have subobjects of type `Iterator` or `Link`.

There are two benefits: *name control* and *access control*. Because the name `Iterator` is nested inside the `LinkedList` class, it is externally known as `LinkedList::Iterator` and cannot conflict with another class called `Iterator`. In Java, this benefit is not as important since Java *packages* give the same kind of name control. Note that the `Link` class is in the *private* part of the `LinkedList` class. It is completely hidden from all other code. For that reason, it is safe to make its data fields public. They can be accessed by the methods of the `LinkedList` class (which has a legitimate need to access them), and they are not visible elsewhere. In Java, this kind of control was not possible until inner classes were introduced.

However, the Java inner classes have an additional feature that makes them richer and more useful than nested classes in C++. An object that comes from an inner class has an implicit reference to the outer class object that instantiated it. Through this pointer, it gains access to the total state of the outer object. You will see the details of the Java mechanism later in this chapter.

Only `static` inner classes do not have this added pointer. They are the Java analog to nested classes in C++.

Using an Inner Class to Access Object State

The syntax for inner classes is somewhat complex. For that reason, we will use a simple but somewhat artificial example to demonstrate the use of inner classes. We will write a program in which a timer controls a bank account. The timer's action listener object adds interest to the account once per second. However, we don't want to use public methods (such as `deposit` or `withdraw`) to manipulate the bank balance because anyone could call those public methods to modify the balance for other purposes. Instead, we will use an *inner class* whose methods can manipulate the bank balance directly.

Here is the outline of the `BankAccount` class:

```
class BankAccount
{
   public BankAccount(double initialBalance) { . . . }
   public void start(double rate) { . . . }

   private double balance;

   private class InterestAdder implements ActionListener
      // an inner class
   {
      . . .
   }
}
```

Note the `InterestAdder` class that is located inside the `BankAccount` class. This does *not* mean that every `BankAccount` has an `InterestAdder` instance field. Of course, we will construct objects of the inner class, but those objects aren't instance fields of the outer class. Instead, they will be local to the methods of the outer class.

The `InterestAdder` class is a *private inner class* inside `BankAccount`. This is a safety mechanism—since only `BankAccount` methods can generate `InterestAdder` objects, we don't have to worry about breaking encapsulation. Only inner classes can be private. Regular classes always have either package or public visibility.

The `InterestAdder` class has a constructor which sets the interest rate that should be applied at each step. Since this inner class implements the `ActionListener` interface, it also has an `actionPerformed` method. That method actually increments the account balance. Here is the inner class in more detail:

```
class BankAccount
{
   public BankAccount(double initialBalance)
   {
      balance = initialBalance;
   }
   . . .
   private double balance;

   private class InterestAdder implements ActionListener
   {
      public InterestAdder(double aRate)
      {
         rate = aRate;
      }

      public void actionPerformed(ActionEvent event) { . . . }

      private double rate;
   }
}
```

The `start` method of the `BankAccount` class constructs an `InterestAdder` object for the given interest rate, makes it the action listener for a timer, and starts the timer.

```
public void start(double rate)
{
   ActionListener adder = new InterestAdder(rate);
   Timer t = new Timer(1000, adder);
   t.start();
}
```

As a result, the `actionPerformed` method of the `InterestAdder` class will be called once per second. Now let's look inside this method more closely:

```
public void actionPerformed(ActionEvent event)
{
   double interest = balance * rate / 100;
   balance += interest;

   NumberFormat formatter
      = NumberFormat.getCurrencyInstance();
   System.out.println("balance="
      + formatter.format(balance));
}
```

The name `rate` refers to the instance field of the `InterestAdder` class, which is not surprising. However, there is no `balance` field in the `InterestAdder` class. Instead, `balance` refers to the field of the `BankAccount` object that created this `InstanceAdder`. This is quite innovative. Traditionally, a method could refer to the data fields of the object invoking the method. An inner class method gets to access both its own data fields *and* those of the outer object creating it.

For this to work, of course, an object of an inner class always gets an implicit reference to the object that created it. (See Figure 6–3.)

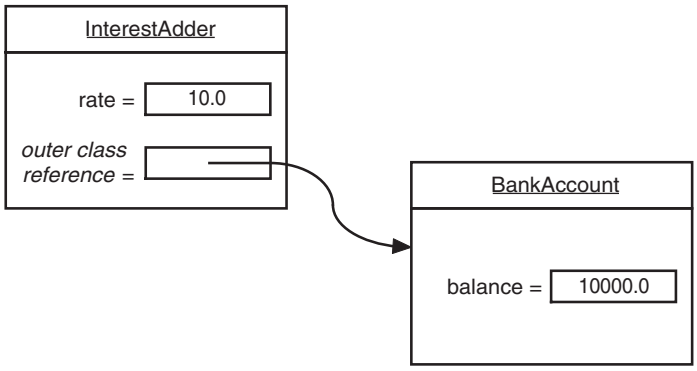

Figure 6–3: An inner class object has a reference to an outer class object

This reference is invisible in the definition of the inner class. However, to illuminate the concept, let us call the reference to the outer object *`outer`*. Then, the `actionPerformed` method is equivalent to the following:

```
public void actionPerformed(ActionEvent event)
{
   double interest = outer.balance * this.rate / 100;
       // "outer" isn't the actual name
   outer.balance += interest;
```

```
      NumberFormat formatter
         = NumberFormat.getCurrencyInstance();
      System.out.println("balance="
         + formatter.format(outer.balance));
   }
```

The outer class reference is set in the constructor. That is, the compiler adds a parameter to the constructor, generating code like this:

```
public InterestAdder(BankAccount account, double aRate)
{
    outer = account;
       // automatically generated code
   rate = aRate;
}
```

Again, please note, *outer* is not a Java keyword. We just use it to illustrate the mechanism involved in an inner class.

When an `InterestAdder` object is constructed in the `start` method, the compiler passes the `this` reference to the current bank account into the constructor:

```
ActionListener adder = new InterestAdder(this, rate);
   // automatically generated code
```

Example 6–4 shows the complete program that tests the inner class. Have another look at the access control. The *Timer* object requires an object of some class that implements the `ActionListener` interface. Had that class been a regular class, then it would have needed to access the bank balance through a public method. As a consequence, the `BankAccount` class would have to provide those methods to all classes, which it might have been reluctant to do. Using an inner class is an improvement. The `InterestAdder` inner class is able to access the bank balance, but no other class has the same privilege.

Example 6–4: InnerClassTest.java

```
1. import java.awt.event.*;
2. import java.text.*;
3. import javax.swing.*;
4.
5. public class InnerClassTest
6. {
7.    public static void main(String[] args)
8.    {
9.       // construct a bank account with initial balance of $10,000
10.      BankAccount account = new BankAccount(10000);
11.      // start accumulating interest at 10%
12.      account.start(10);
13.
14.      // keep program running until user selects "Ok"
15.      JOptionPane.showMessageDialog(null, "Quit program?");
16.      System.exit(0);
17.   }
18. }
19.
20. class BankAccount
21. {
22.   /**
23.      Constructs a bank account with an initial balance
24.      @param initialBalance the initial balance
25.   */
```


```
26.    public BankAccount(double initialBalance)
27.    {
28.       balance = initialBalance;
29.    }
30.
31.    /**
32.       Starts a simulation in which interest is added once per
33.       second
34.       @param rate the interest rate in percent
35.    */
36.    public void start(double rate)
37.    {
38.       ActionListener adder = new InterestAdder(rate);
39.       Timer t = new Timer(1000, adder);
40.       t.start();
41.    }
42.
43.    private double balance;
44.
45.    /**
46.       This class adds the interest to the bank account.
47.       The actionPerformed method is called by the timer.
48.    */
49.    private class InterestAdder implements ActionListener
50.    {
51.       public InterestAdder(double aRate)
52.       {
53.          rate = aRate;
54.       }
55.
56.       public void actionPerformed(ActionEvent event)
57.       {
58.          // update interest
59.          double interest = balance * rate / 100;
60.          balance += interest;
61.
62.          // print out current balance
63.          NumberFormat formatter
64.             = NumberFormat.getCurrencyInstance();
65.          System.out.println("balance="
66.             + formatter.format(balance));
67.       }
68.
69.       private double rate;
70.    }
71. }
```

Special Syntax Rules for Inner Classes

In the preceding section, we explained the outer class reference of an inner class by calling it *outer*. Actually, the proper syntax for the outer reference is a bit more complex. The expression

```
OuterClass.this
```

denotes the outer class reference. For example, you can write the `actionPerformed` method of the `InterestAdder` inner class as

```
public void actionPerformed(ActionEvent event)
{
   double interest = BankAccount.this.balance * this.rate / 100;
```

```
      BankAccount.this.balance += interest;
      . . .
   }
```

Conversely, you can write the inner object constructor more explicitly, using the syntax:

```
outerObject.new InnerClass(construction parameters)
```

For example,

```
ActionListener adder = this.new InterestAdder(rate);
```

Here, the outer class reference of the newly constructed `InterestAdder` object is set to the `this` reference of the method that creates the inner class object. This is the most common case. As always, the `this.` qualifier is redundant. However, it is also possible to set the outer class reference to another object by explicitly naming it. For example, if `InterestAdder` was a public inner class, you could construct an `InterestAdder` for any bank account:

```
BankAccount mySavings = new BankAccount(10000);
BankAccount.InterestAdder adder
   = mySavings.new InterestAdder(10);
```

Note that you refer to an inner class as

```
OuterClass.InnerClass
```

when it occurs outside the scope of the outer class. For example, if `InterestAdder` had been a public class, you could have referred to it as `BankAccount.InterestAdder` elsewhere in your program.

Are Inner Classes Useful? Are They Actually Necessary? Are They Secure?

Inner classes are a major addition to the language. Java started out with the goal of being simpler than C++. But inner classes are anything but simple. The syntax is complex. (It will get more complex as we study anonymous inner classes later in this chapter.) It is not obvious how inner classes interact with other features of the language, such as access control and security.

Has Java started down the road to ruin that has afflicted so many other languages, by adding a feature that was elegant and interesting rather than needed?

While we won't try to answer this question completely, it is worth noting that inner classes are a phenomenon of the *compiler,* not the virtual machine. Inner classes are translated into regular class files with `$` (dollar signs) delimiting outer and inner class names, and the virtual machine does not have any special knowledge about them.

For example, the `InterestAdder` class inside the `BankAccount` class is translated to a class file `BankAccount$InterestAdder.class`. To see this at work, try out the following experiment: run the `ReflectionTest` program of Chapter 5, and give it the class `BankAccount$InterestAdder` to reflect upon. You will get the following printout:

```
class BankAccount$InterestAdder
{
   public BankAccount$InterestAdder(BankAccount, double);

   public void actionPerformed(java.awt.event.ActionEvent);

   private double rate;
   private final BankAccount this$0;
}
```

If you use Unix, remember to escape the `$` character if you supply the class name on the command line. That is, run the `ReflectionTest` program as

```
java ReflectionTest 'BankAccount$InterestAdder'
```

You can plainly see that the compiler has generated an additional instance field, `this$0`, for the reference to the outer class. (The name `this$0` is synthesized by the compiler—you cannot refer to it in your code.) You can also see the added parameter for the constructor.

If the compiler can do this transformation, couldn't you simply program the same mechanism by hand? Let's try it. We would make `InterestAdder` a regular class, outside the `BankAccount` class. When constructing an `InterestAdder` object, we pass it the `this` reference of the object that is creating it.

```
class BankAccount
{  . . .

   public void start(double rate)
   {
      ActionListener adder = new InterestAdder(this, rate);
      Timer t = new Timer(1000, adder);
      t.start();
   }
}

class InterestAdder implements ActionListener
{
   public InterestAdder(BankAccount account, double aRate)
   {
       outer = account;
      rate = aRate;
   }
   . . .
   private BankAccount outer;
   private double rate;
 }
```

Now let us look at the `actionPerformed` method. It needs to access `outer.balance`.

```
public void actionPerformed(ActionEvent event)
{
   double interest = outer.balance * rate / 100; // ERROR
   outer.balance += interest; // ERROR
   . . .
}
```

Here we run into a problem. The inner class can access the private data of the outer class, but our external `InterestAdder` class cannot.

Thus, inner classes are genuinely more powerful than regular classes, since they have more access privileges.

You may well wonder how inner classes manage to acquire those added access privileges, since inner classes are translated to regular classes with funny names—the virtual machine knows nothing at all about them. To solve this mystery, let's again use the `ReflectionTest` program to spy on the `BankAccount` class:

```
class BankAccount
{
```

```
   public BankAccount(double);

   static double access$000(BankAccount);
   public void start(double);
   static double access$018(BankAccount, double);

   private double balance;
}
```

Notice the static `access$000` and `access$018` methods that the compiler added to the outer class. The inner class methods call those methods. For example, the statement

```
balance += interest
```

in the `actionPerformed` method of the `InterestAdder` class effectively makes the following call:

```
access$018(outer, access$000(outer) + interest);
```

Is this a security risk? You bet it is. It is an easy matter for someone else to invoke the `access$000` method to read the private `balance` field or, even worse, to call the `access$018` method to set it. The Java language standard reserves `$` characters in variable and method names for system usage. However, for those hackers who are familiar with the structure of class files, it is an easy (if tedious) matter to produce a class file with virtual machine instructions to call that method. Of course, such a class file would need to be generated manually (for example, with a hex editor). Because the secret access methods have package visibility, the attack code would need to be placed inside the same package as the class under attack.

To summarize, if an inner class accesses a private data field, then it is possible to access that data field through other classes that are added to the package of the outer class, but to do so requires skill and determination. A programmer cannot accidentally obtain access but must intentionally build or modify a class file for that purpose.

Local Inner Classes

If you look carefully at the code of the `BankAccount` example, you will find that you need the name of the type `InterestAdder` only once: when you create an object of that type in the `start` method.

When you have a situation like this, you can define the class *locally in a single method*.

```
public void start(double rate)
{
   class InterestAdder implements ActionListener
   {
      public InterestAdder(double aRate)
      {
         rate = aRate;
      }

      public void actionPerformed(ActionEvent event)
      {
         double interest = balance * rate / 100;
         balance += interest;

         NumberFormat formatter
            = NumberFormat.getCurrencyInstance();
         System.out.println("balance="
            + formatter.format(balance));
      }
```


```
      private double rate;
   }

   ActionListener adder = new InterestAdder(rate);
   Timer t = new Timer(1000, adder);
   t.start();
}
```

Local classes are never declared with an access specifier (that is, `public` or `private`). Their scope is always restricted to the block in which they are declared.

Local classes have a great advantage—they are completely hidden from the outside world—not even other code in the `BankAccount` class can access them. No method except `start` has any knowledge of the `InterestAdder` class.

Local classes have another advantage over other inner classes. Not only can they access the fields of their outer classes, they can even access local variables! However, those local variables must be declared `final`. Here is a typical example.

```
public void start(final double rate)
{
   class InterestAdder implements ActionListener
   {
      public void actionPerformed(ActionEvent event)
      {
         double interest = balance * rate / 100;
         balance += interest;

         NumberFormat formatter
            = NumberFormat.getCurrencyInstance();
         System.out.println("balance="
            + formatter.format(balance));
      }
  }

   ActionListener adder = new InterestAdder();
   Timer t = new Timer(1000, adder);
   t.start();
}
```

Note that the `InterestAdder` class no longer needs to store a `rate` instance variable. It simply refers to the parameter variable of the method that contains the class definition.

Maybe this should not be so surprising. The line

```
double interest = balance * rate / 100;
```

is, after all, ultimately inside the `start` method, so why shouldn't it have access to the value of the `rate` variable?

To see why there is a subtle issue here, let's consider the flow of control more closely.

1. The `start` method is called.
2. The object variable `adder` is initialized via a call to the constructor of the inner class `InterestAdder`.
3. The `adder` reference is passed to the `Timer` constructor, the timer is started, and the `start` method exits. At this point, the `rate` parameter variable of the `start` method no longer exists.
4. A second later, the `actionPerformed` method calls `double interest = balance * rate / 100;`.

For the code in the `actionPerformed` method to work, the `InterestAdder` class must have made a copy of the `rate` field before it went away as a local variable of the `start` method. That is indeed exactly what happens. In our example, the compiler synthesizes the name `BankAccount$1$InterestAdder` for the local inner class. If you use the `ReflectionTest` program again to spy on the `BankAccount$1$InterestAdder` class, you get the following output:

```
class BankAccount$1$InterestAdder
{
   BankAccount$1$InterestAdder(BankAccount, double);

   public void actionPerformed(java.awt.event.ActionEvent);

   private final double val$rate;
   private final BankAccount this$0;
}
```

Note the extra `double` parameter to the constructor and the `val$rate` instance variable. When an object is created, the value `rate` is passed into the constructor and stored in the `val$rate` field. This sounds like an extraordinary amount of trouble for the implementors of the compiler. The compiler must detect access of local variables, make matching data fields for each one of them, and copy the local variables into the constructor so that the data fields can be initialized as copies of them.

From the programmer's point of view, however, local variable access is quite pleasant. It makes your inner classes simpler by reducing the instance fields that you need to program explicitly.

As we already mentioned, the methods of a local class can refer only to local variables that are declared `final`. For that reason, the `rate` parameter was declared `final` in our example. A local variable that is declared `final` cannot be modified. Thus, it is guaranteed that the local variable and the copy that is made inside the local class always have the same value.

You have seen `final` variables used for constants, such as

```
public static final double SPEED_LIMIT = 55;
```

The `final` keyword can be applied to local variables, instance variables, and static variables. In all cases it means the same thing: You can assign to this variable *once* after it has been created. Afterwards, you cannot change the value—it is final.

However, you don't have to initialize a `final` variable when you define it. For example, the `final` parameter variable `rate` is initialized once after its creation, when the `start` method is called. (If the method is called multiple times, each call has its own newly created `rate` parameter.) The `val$rate` instance variable that you can see in the `BankAccount$1$InterestAdder` inner class is set once, in the inner class constructor. A final variable that isn't initialized when it is defined is often called a *blank final* variable.

Anonymous inner classes

When using local inner classes, you can often go a step further. If you want to make only a single object of this class, you don't even need to give the class a name. Such a class is called *anonymous inner class.*

```
public void start(final double rate)
{
   ActionListener adder = new
      ActionListener()
      {
```


```
            public void actionPerformed(ActionEvent event)
            {
               double interest = balance * rate / 100;
               balance += interest;

               NumberFormat formatter
                  = NumberFormat.getCurrencyInstance();
               System.out.println("balance="
                  + formatter.format(balance));
            }
         };
      Timer t = new Timer(1000, adder);
      t.start();
   }
```

This is a very cryptic syntax indeed. What it means is:

Create a new object of a class that implements the `ActionListener` interface, where the required method `actionPerformed` is the one defined inside the braces `{ }`.

Any parameters used to construct the object are given inside the parentheses `( )` following the supertype name. In general, the syntax is

```
new SuperType(construction parameters)
{
   inner class methods and data
}
```

Here, *`SuperType`* can be an interface, such as `ActionListener`; then, the inner class *implements* that interface. Or, *`SuperType`* can be a class; then, the inner class *extends* that class.

An anonymous inner class cannot have constructors because the name of a constructor must be the same as the name of a class, and the class has no name. Instead, the construction parameters are given to the *superclass* constructor. In particular, whenever an inner class implements an interface, it cannot have any construction parameters. Nevertheless, you must supply a set of parentheses as in:

```
new InterfaceType() { methods and data }
```

You have to look very carefully to see the difference between the construction of a new object of a class and the construction of an object of an anonymous inner class extending that class. If the closing parenthesis of the construction parameter list is followed by an opening brace, then an anonymous inner class is being defined.

```
Person queen = new Person("Mary");
   // a Person object
Person count = new Person("Dracula") { ... };
   // an object of an inner class extending Person
```

Are anonymous inner classes a great idea or are they a great way of writing obfuscated code? Probably a bit of both. When the code for an inner class is short, just a few lines of simple code, then they can save typing time, but it is exactly timesaving features like this that lead you down the slippery slope to "Obfuscated Java Code Contests."

It is a shame that the designers of Java did not try to improve the syntax of anonymous inner classes, since, generally, Java syntax is a great improvement over C++. The designers of the inner class feature could have helped the human reader with a syntax such as:

```
Person count = new class extends Person("Dracula") { ... };
   // not the actual Java syntax
```

But they didn't. Because many programmers find code with too many anonymous inner classes hard to read, we recommend restraint when using them.

Example 6–5 contains the complete source code for the bank account program with an anonymous inner class. If you compare this program with Example 6–4, you will find that in this case the solution with the anonymous inner class is quite a bit shorter, and, hopefully, with a bit of practice, as easy to comprehend.

Example 6–5: AnonymousInnerClassTest.java

```
1. import java.awt.event.*;
2. import java.text.*;
3. import javax.swing.*;
4.
5. public class AnonymousInnerClassTest
6. {
7.    public static void main(String[] args)
8.    {
9.       // construct a bank account with initial balance of $10,000
10.      BankAccount account = new BankAccount(10000);
11.      // start accumulating interest at 10%
12.      account.start(10);
13.
14.      // keep program running until user selects "Ok"
15.      JOptionPane.showMessageDialog(null, "Quit program?");
16.      System.exit(0);
17.   }
18. }
19.
20. class BankAccount
21. {
22.   /**
23.      Constructs a bank account with an initial balance
24.      @param initialBalance the initial balance
25.   */
26.   public BankAccount(double initialBalance)
27.   {
28.      balance = initialBalance;
29.   }
30.
31.   /**
32.      Starts a simulation in which interest is added once per
33.      second
34.      @param rate the interest rate in percent
35.   */
36.   public void start(final double rate)
37.   {
38.      ActionListener adder = new
39.         ActionListener()
40.         {
41.            public void actionPerformed(ActionEvent event)
42.            {
43.               // update interest
44.               double interest = balance * rate / 100;
45.               balance += interest;
46.
47.               // print out current balance
48.               NumberFormat formatter
```


```
49.                     = NumberFormat.getCurrencyInstance();
50.                  System.out.println("balance="
51.                     + formatter.format(balance));
52.               }
53.            };
54.
55.         Timer t = new Timer(1000, adder);
56.         t.start();
57.      }
58.
59.      private double balance;
60. }
```

Static Inner Classes

Occasionally, you want to use an inner class simply to hide one class inside another, but you don't need the inner class to have a reference to the outer class object. You can suppress the generation of that reference by declaring the inner class `static`.

Here is a typical example of where you would want to do this. Consider the task of computing the minimum and maximum value in an array. Of course, you write one function to compute the minimum and another function to compute the maximum. When you call both functions, then the array is traversed twice. It would be more efficient to traverse the array only once, computing both the minimum and the maximum simultaneously.

```
double min = d[0];
double max = d[0];
for (int i = 1; i < d.length; i++)
{  if (min > d[i]) min = d[i];
   if (max < d[i]) max = d[i];
}
```

However, the function must return two numbers. We can achieve that by defining a class `Pair` that holds two values:

```
class Pair
{  public Pair(double f, double s)
   {  first = f;
      second = s;
   }
   public double getFirst()
   {  return first;
   }
   public double getSecond()
   {  return second;
   }

   private double first;
   private double second;
}
```

The `minmax` function can then return an object of type `Pair`.

```
class ArrayAlg
{  public static Pair minmax(double[] d)
   {  . . .
      return new Pair(min, max);
   }
}
```

The caller of the function then uses the `getFirst` and `getSecond` methods to retrieve the answers:

```
Pair p = ArrayAlg.minmax(d);
System.out.println("min = " + p.getFirst());
System.out.println("max = " + p.getSecond());
```

Of course, the name `Pair` is an exceedingly common name, and in a large project, it is quite possible that some other programmer had the same bright idea, except that the other programmer made a `Pair` class that contains a pair of strings. We can solve this potential name clash by making `Pair` a public inner class inside `ArrayAlg`. Then the class will be known to the public as `ArrayAlg.Pair`:

```
ArrayAlg.Pair p = ArrayAlg.minmax(d);
```

However, unlike the inner classes that we used in previous examples, we do not want to have a reference to any other object inside a `Pair` object. That reference can be suppressed by declaring the inner class `static`:

```
class ArrayAlg
{
   public static class Pair
   {
     . . .
   }
   . . .
}
```

Of course, only inner classes can be declared static. A static inner class is exactly like any other inner class, except that an object of a static inner class does not have a reference to the outer class object that generated it. In our example, we must use a static inner class because the inner class object is constructed inside a static method:

```
public static Pair minmax(double[] d)
{
   . . .
   return new Pair(min, max);
}
```

Had the `Pair` class not been declared as `static`, the compiler would have complained that there was no implicit object of type `ArrayAlg` available to initialize the inner class object.

You use a static inner class whenever the inner class does not need to access an outer class object. Some programmers use the term *nested class* to describe static inner classes.

Example 6–6 contains the complete source code of the `ArrayAlg` class and the nested `Pair` class.

Example 6–6: StaticInnerClassTest.java

```
1. public class StaticInnerClassTest
2. {
3.    public static void main(String[] args)
4.    {
5.       double[] d = new double[20];
6.       for (int i = 0; i < d.length; i++)
7.          d[i] = 100 * Math.random();
8.       ArrayAlg.Pair p = ArrayAlg.minmax(d);
9.       System.out.println("min = " + p.getFirst());
10.      System.out.println("max = " + p.getSecond());
11.   }
12. }
13.
```


```
14. class ArrayAlg
15. {
16.    /**
17.       A pair of floating point numbers
18.    */
19.    public static class Pair
20.    {
21.       /**
22.           Constructs a pair from two floating point numbers
23.           @param f the first number
24.           @param s the second number
25.       */
26.       public Pair(double f, double s)
27.       {
28.          first = f;
29.          second = s;
30.       }
31.
32.       /**
33.          Returns the first number of the pair
34.          @return the first number
35.       */
36.       public double getFirst()
37.       {
38.          return first;
39.       }
40.
41.       /**
42.          Returns the second number of the pair
43.          @return the second number
44.       */
45.       public double getSecond()
46.       {
47.          return second;
48.       }
49.
50.       private double first;
51.       private double second;
52.    }
53.
54.    /**
55.       Computes both the minimum and the maximum of an array
56.       @param a an array of floating point numbers
57.       @return a pair whose first element is the minimum and whose
58.       second element is the maximum
59.    */
60.    public static Pair minmax(double[] d)
61.    {
62.       if (d.length == 0) return new Pair(0, 0);
63.       double min = d[0];
64.       double max = d[0];
65.       for (int i = 1; i < d.length; i++)
66.       {
67.          if (min > d[i]) min = d[i];
68.          if (max < d[i]) max = d[i];
69.       }
```

```
70.         return new Pair(min, max);
71.     }
72. }
```

Proxies

In the final section of this chapter, we will discuss *proxies*, a new feature that became available with version 1.3 of the Java SDK. You use a proxy to create new classes at runtime that implement a given set of interfaces. Proxies are only necessary when you don't yet know at compile time which interfaces you need to implement. This is not a common situation for application programmers. However, for certain system programming applications the flexibility that proxies offer can be very important. By using proxies, you can often avoid the mechanical generation and compilation of "stub" code.

In versions 1.2 and below of the Java SDK, you encounter stub code in a number of situations. When you use *remote method invocation* (RMI), a special utility called `rmic` produces stub classes that you need to add to your program. (See Chapter 4 of Volume 2 for more information on RMI.) And when you use the *bean box*, stub classes are produced and compiled on the fly when you connect beans to each other. (See Chapter 7 of Volume 2 for more information on Java beans.) It is expected that these mechanisms will be updated soon to take advantage of the proxy capability.

Suppose you have an array of `Class` objects representing interfaces (maybe only containing a single interface), whose exact nature you may not know at compile time. Now you want to construct an object of a class that implements these interfaces. This is a difficult problem. If a `Class` object represents an actual class, then you can simply use the `newInstance` method or use reflection to find a constructor of that class. But you can't instantiate an interface. And you can't define new classes in a running program.

To overcome this problem, some programs—such as the BeanBox in early versions of the Bean Development Kit—generate code, place it into a file, invoke the compiler and then load the resulting class file. Naturally, this is slow, and it also requires deployment of the compiler together with the program. The *proxy* mechanism is a better solution. The proxy class can create brand-new classes at runtime. Such a proxy class implements the interfaces that you specify. In particular, the proxy class has the following methods:

- All methods required by the specified interfaces;
- All methods defined in the `Object` class (`toString`, `equals`, and so on).

However, you cannot define new code for these methods at runtime. Instead, you must supply an *invocation handler*. An invocation handler is an object of any class that implements the `InvocationHandler` interface. That interface has a single method:

```
Object invoke(Object proxy, Method method, Object[] args)
```

Whenever a method is called on the proxy object, the `invoke` method of the invocation handler gets called, with the `Method` object and parameters of the original call. The invocation handler must then figure out how to handle the call.

To create a proxy object, you use the `newProxyInstance` method of the `Proxy` class. The method has three parameters:

1. A *class loader*. As part of the Java security model, it is possible to use different class loaders for system classes, classes that are downloaded from the Internet, and so on. We will discuss class loaders in Volume 2. For now, we will specify `null` to use the default class loader.
2. An array of `Class` objects, one for each interface to be implemented.

3. An invocation handler.

There are two remaining questions. How do we define the handler? And what can we do with the resulting proxy object? The answers depend, of course, on the problem that we want to solve with the proxy mechanism. Proxies can be used for many purposes, such as:

- Routing method calls to remote servers;
- Associating user interface events with actions in a running program;
- Tracing method calls for debugging purposes.

In our example program, we will use proxies and invocation handlers to trace method calls. We define a `TraceHandler` wrapper class that stores a wrapped object. Its `invoke` method simply prints out the name and parameters of the method to be called, and then calls the method with the wrapped object as the implicit parameter.

```
class TraceHandler implements InvocationHandler
{
   public TraceHandler(Object t)
   {
      target = t;
   }

   public Object invoke(Object proxy, Method m, Object[] args)
      throws Throwable
   {
      // print method name and parameters
      . . .
      // invoke actual method
      return m.invoke(target, args);
   }

   private Object target;
}
```

Here is how you construct a proxy object that causes the tracing behavior whenever one of its methods is called.

```
Object value = . . .;
// construct wrapper
InvocationHandler handler = new TraceHandler(value);
// construct proxy for all interfaces
Class[] interfaces = value.getClass().getInterfaces();
Object proxy = Proxy.newProxyInstance(null, interfaces, handler);
```

Now, whenever a method is called on `proxy`, the method name and parameters are printed out, and then the method is invoked on `value`.

In the program shown in Example 6–7, we use proxy objects to trace a binary search. We fill an array with proxies to the integers 1 . . . 1000. Then we invoke the `binarySearch` method of the `Arrays` class to search for a random integer in the array. Finally, we print out the matching element.

```
Object[] elements = new Object[1000];

// fill elements with proxies for the integers 1 ... 1000
for (int i = 0; i < elements.length; i++)
{
   Integer value = new Integer(i + 1);
   elements[i] = . . .; // proxy for value;
}
```

```
// construct a random integer
Random generator = new Random();
int r = generator.nextInt(elements.length);
Integer key = new Integer(r + 1);

// search for the key
int result = Arrays.binarySearch(elements, key);

// print match if found
if (result >= 0)
   System.out.println(elements[result]);
```

The `Integer` class implements the `Comparable` interface. The proxy objects belong to a class that is defined at runtime. (It has a name such as `$Proxy0`.) That class also implements the `Comparable` interface. However, its `compareTo` method calls the `invoke` method of the proxy object's handler.

The `binarySearch` method makes calls like this:

```
if (elements[i].compareTo(key) < 0) . . .
```

Because we filled the array with proxy objects, the `compareTo` calls call the `invoke` method of the `TraceHandler` class. That method prints out the method name and parameters and then invokes `compareTo` on the wrapped `Integer` object.

Finally, at the end of the sample program, we call:

```
System.out.println(elements[result]);
```

The `println` method calls `toString` on the proxy object, and that call is also redirected to the invocation handler.

Here is the complete trace of a program run:

```
500.compareTo(288)
250.compareTo(288)
375.compareTo(288)
312.compareTo(288)
281.compareTo(288)
296.compareTo(288)
288.compareTo(288)
288.toString()
```

You can see how the binary search algorithm homes in on the key, by cutting the search interval in half in every step.

Example 6–7: ProxyTest.java

```
1. import java.lang.reflect.*;
2. import java.util.*;
3.
4. public class ProxyTest
5. {
6.    public static void main(String[] args)
7.    {
8.       Object[] elements = new Object[1000];
9.
10.      // fill elements with proxies for the integers 1 ... 1000
11.      for (int i = 0; i < elements.length; i++)
12.      {
13.         Integer value = new Integer(i + 1);
14.         Class[] interfaces = value.getClass().getInterfaces();
15.         InvocationHandler handler = new TraceHandler(value);
16.         Object proxy = Proxy.newProxyInstance(null,
17.            interfaces, handler);
18.         elements[i] = proxy;
19.      }
```


```
20.
21.       // construct a random integer
22.       Random generator = new Random();
23.       int r = generator.nextInt(elements.length);
24.       Integer key = new Integer(r + 1);
25.
26.       // search for the key
27.       int result = Arrays.binarySearch(elements, key);
28.
29.       // print match if found
30.       if (result >= 0)
31.          System.out.println(elements[result]);
32.    }
33. }
34.
35. /**
36.    An invocation handler that prints out the method name
37.    and parameters, then invokes the original method
38. */
39. class TraceHandler implements InvocationHandler
40. {
41.    /**
42.       Constructs a TraceHandler
43.       @param t the implicit parameter of the method call
44.    */
45.    public TraceHandler(Object t)
46.    {
47.       target = t;
48.    }
49.
50.    public Object invoke(Object proxy, Method m, Object[] args)
51.       throws Throwable
52.    {
53.       // print implicit argument
54.       System.out.print(target);
55.       // print method name
56.       System.out.print("." + m.getName() + "(");
57.       // print explicit arguments
58.       if (args != null)
59.       {
60.          for (int i = 0; i < args.length; i++)
61.          {
62.             System.out.print(args[i]);
63.             if (i < args.length - 1)
64.                System.out.print(", ");
65.          }
66.       }
67.       System.out.println(")");
68.
69.       // invoke actual method
70.       return m.invoke(target, args);
71.    }
72.
73.    private Object target;
74. }
```

Properties of Proxy Classes

Now that you have seen proxy classes in action, we want to go over some of their properties. Remember that proxy classes are created on the fly, in a running program. However, once they are created, they are regular classes, just like any other classes in the virtual machine.

All proxy classes extend the class `Proxy`. A proxy class has only one instance variable—the invocation handler which is defined in the `Proxy` superclass. Any additional data that is required to carry out the proxy objects' tasks must be stored in the invocation handler. For example, when we proxied `Comparable` objects in the program shown in Example 6–7, the `TraceHandler` wrapped the actual objects.

All proxy classes override the `toString`, `equals`, and *`hashCode`* methods of the `Object` class. Like all proxy methods, these methods simply call `invoke` on the invocation handler. The other methods of the `Object` class (such as `clone` and `getClass`) are not redefined.

The names of proxy classes are not defined. The `Proxy` class in the Java 2 SDK generates class names that begin with the string `$Proxy`.

There is only one proxy class for a particular class loader and ordered set of interfaces. That is, if you call the `newProxyInstance` method twice with the same class loader and interface array, then you get two objects of the same class. You can also obtain that class with the `getProxyClass` method:

```
Class proxyClass = Proxy.getProxyClass(null, interfaces);
```

A proxy class is always `public` and `final`. If all interfaces that the proxy class implements are `public`, then the proxy class does not belong to any particular package. Otherwise, all non-public interfaces must belong to the same package, and then the proxy class also belongs to that package.

You can test whether a particular `Class` object represents a proxy class, by calling the `isProxyClass` method of the `Proxy` class.

This ends our final chapter on the fundamentals of the Java programming language. Interfaces and inner classes are concepts that you will encounter frequently. However, as we already mentioned, proxies are an advanced technique that is of interest mainly to tool builders, not application programmers. You are now ready to go on to learn about graphics and user interfaces, starting with Chapter 7.

java.lang.reflect.InvocationHandler 1.3

- `Object invoke(Object proxy, Method method, Object[] args)`
  Define this method to contain the action that you want carried out whenever a method was invoked on the proxy object.

java.lang.reflect.Proxy 1.3

- `static Class getProxyClass(ClassLoader loader, Class[] interfaces)`
  Returns the proxy class that implements the given interfaces.
- `static Object newProxyInstance(ClassLoader loader, Class[] interfaces, InvocationHandler handler)`
  Constructs a new instance of the proxy class that implements the given interfaces. All methods call the `invoke` method of the given handler object.
- `static boolean isProxyClass(Class c)`
  Returns `true` if `c` is a proxy class.