Chapter 13

Interfaces and Inner Classes

Interfaces

• An interface serves a function similar to a base class, though it is not a base class
  – Some languages allow one class to be derived from two or more different base classes
  – This *multiple inheritance* is not allowed in Java
  – Instead, Java’s way of approximating multiple inheritance is through interfaces

• An interface is something like an extreme case of an abstract class
  – However, an interface is not a class
  – It is a type that can be satisfied by any class that implements the interface

• The syntax for defining an interface is similar to that of defining a class
  – Except the word *interface* is used in place of *class*

• An interface specifies a set of methods that any class that implements the interface must have
  – It contains method headings and constant definitions only
  – It contains no instance variables nor any complete method definitions

• An interface and all of its method headings should be declared public
  – They cannot be given private, protected, or package access

• When a class implements an interface, it must make all the methods in the interface public

• Because an interface is a type, a method may be written with a parameter of an interface type
  – That parameter will accept as an argument any class that implements the interface
The **Ordered** Interface

```java
public interface Ordered {
    public boolean precedes(Object other);
    public boolean follows(Object other);
}
```

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.

**Interfaces**

- To *implement an interface*, a concrete class must do two things:
  1. It must include the phrase `implements Interface_Name` at the start of the class definition
  2. The class must implement all the method headings listed in the definition(s) of the interface(s)

- Note the use of `Object` as the parameter type in the following examples

---

**Implementation of an Interface**

```java
public class OrderableEmployee extends HourlyEmployee implements Ordered {
    public boolean precedes(Object other) {
        if (other == null)
            return false;
        else if (other instanceof HourlyEmployee)
            return false;
        else {
            OrderedHourlyEmployee otherOrderedHourlyEmployee = (OrderedHourlyEmployee) other;
            return getPay() < otherOrderedHourlyEmployee.getPay();
        }
    }
}
```

---

**Implementation of an Interface (continued)**

```java
public boolean follows(Object other) {
    if (other == null)
        return false;
    else if (other instanceof OrderedHourlyEmployee)
        return false;
    else {
        OrderedHourlyEmployee otherOrderedHourlyEmployee = (OrderedHourlyEmployee) other;
        return (otherOrderedHourlyEmployee.precedes(this));
    }
}
```
Abstract Classes Implementing Interfaces

- Abstract classes may implement one or more interfaces
  - Any method headings given in the interface that are not given definitions are made into abstract methods
- A concrete class must give definitions for all the method headings given in the abstract class and the interface

An Abstract Class Implementing an Interface

```java
public abstract class MyAbstractClass implements Ordered {
    int number;
    char grade;
    public boolean precedes(Object other) {
        if (other == null)
            return false;
        else if (other instanceof HourlyEmployee)
            return false;
        else
            return this.number < other.number;
    }
    public abstract boolean follows(Object other);
}
```

Derived Interfaces

- Like classes, an interface may be derived from a base interface
  - This is called extending the interface
  - The derived interface must include the phrase `extends BaseInterfaceName`
- A concrete class that implements a derived interface must have definitions for any methods in the derived interface as well as any methods in the base interface

Extending an Interface

```java
public interface ShowablyOrdered extends Ordered {
    /**
     * Outputs an object of the class that precedes the calling object.
     *
     * @return true if there is an object that precedes this object.
     */
    public boolean showOneWhoPrecedes();
}
```
Pitfall: Interface Semantics Are Not Enforced

- When a class implements an interface, the compiler and runtime system check the syntax of the interface and its implementation
  - However, neither checks that the body of an interface is consistent with its intended meaning
- Required semantics for an interface are normally added to the documentation for an interface
  - It then becomes the responsibility of each programmer implementing the interface to follow the semantics
- If the method body does not satisfy the specified semantics, then software written for classes that implement the interface may not work correctly

The Comparable Interface

- The Comparable interface is in the java.lang package, and so is automatically available to any program
- It has only the following method heading that must be implemented:
  ```java
  public int compareTo(Object other);
  ```
- It is the programmer's responsibility to follow the semantics of the Comparable interface when implementing it

The Comparable Interface Semantics

- The method compareTo must return
  - A negative number if the calling object "comes before" the parameter other
  - A zero if the calling object "equals" the parameter other
  - A positive number if the calling object "comes after" the parameter other
- If the parameter other is not of the same type as the class being defined, then a ClassCastException should be thrown
The **Comparable** Interface Semantics

- Almost any reasonable notion of "comes before" is acceptable
  - In particular, all of the standard less-than relations on numbers and lexicographic ordering on strings are suitable
- The relationship "comes after" is just the reverse of "comes before"

- Other orderings may be considered, as long as they are a total ordering
- Such an ordering must satisfy the following rules:
  - (Irreflexivity) For no object o does o come before o
  - (Trichotomy) For any two object 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 method if possible, but this is not absolutely required

Using the **Comparable** Interface

- The following example reworks the SelectionSort class from Chapter 6
- The new version, GeneralizedSelectionSort, includes a method that can sort any partially filled array whose base type implements the Comparable interface
  - It contains appropriate indexOfSmallest and interchange methods as well
- Note: Both the Double and String classes implement the Comparable interface
  - Interfaces apply to classes only
  - A primitive type (e.g., double) cannot implement an interface

```java
public class GeneralizedSelectionSort
{
    /**
    * Precondition: numberOfUsed == a.length.
    * The first numberOfUsed indexed variables have values.
    * Action: Sorts a so that a[0], a[1], ..., a[numberOfUsed - 1] are in increasing order by the compareTo method.
    */
    public static void sort(Comparable[] a, int numberOfUsed)
    {
        int index, indexOfNextSmallest;
        for (index = 0; index < numberOfUsed - 1; index++)
        {
            //Place the correct value in a[index]:
            int indexOfNextSmallest = indexOfSmallest(index, a, numberOfUsed);
            interchange(index, indexOfNextSmallest, a);
            //a[0], a[1], ..., a[index] are correctly ordered and there are
            //the smallest of the original array elements. The remaining
            //positions contain the rest of the original array elements.
        }
    }
}
```

GeneralizedSelectionSort class: sort Method
Genera\textit{lizedSelectionSort} class: \textit{sort} Method

\begin{verbatim}
Display 13.5 Sorting Method for Array of Comparable (Part 1 of 2) (continued)

21 //
22 Returns the index of the smallest value among
23 [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 indexMin = startIndex;
30     int index;
31     for (index = startIndex + 1; index < numberUsed; index++)
32         if (a[index].compareTo(min) < 0) //a[index] is less than min
33             
34                 min = a[index];
35                 indexMin = index;
36             //min is smallest of a[startIndex] through a[index]
37         }
38     return indexMin;
39 }
\end{verbatim}

\textit{GeneralizedSelectionSort} class: \textit{interchange} Method

\begin{verbatim}
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 1, int j, Comparable[] a)
{
    Comparable temp;
    temp = a[i];
    a[i] = a[j];
    a[j] = temp; //original value of a[1]
}
\end{verbatim}

\textbf{Sorting Arrays of Comparable}

\begin{verbatim}
Display 13.6 Sorting arrays of Comparable (Part 1 of 2)

/**
 * Demonstrates sorting arrays for classes that implement the 
 * Comparable interface.
 */
\end{verbatim}

\begin{verbatim}
public class ComparableDemo 
{
    public static void main(String[] args)
    {
        Double[] d = new Double[10];
        for (int i = 0; i < d.length; i++)
            d[i] = new Double(i * 1.5 - 1); 
        System.out.println("Before sorting:");
        for (int i = 0; i < numberUsed; i++)
            System.out.println(d[i].doubleValue() + ", ");
        System.out.println();
        GeneralizedSelectionSort.sort(d, d.length);
        System.out.println("After sorting:");
        for (i = 0; i < d.length; i++)
            System.out.println(d[i].doubleValue() + ", ");
        System.out.println();
    }
\end{verbatim}

\textbf{Sorting Arrays of Comparable}

\begin{verbatim}
Display 13.6 Sorting arrays of Comparable (Part 2 of 2)

String[] a = new String[10];
a[0] = "000";
a[1] = "cct";
a[2] = "cornish game hen";
int numberUsed = 3;
System.out.println("Before sorting:");
for (i = 0; i < numberUsed; i++)
    System.out.println(a[i] + ", ");
System.out.println();
GeneralizedSelectionSort.sort(a, numberUsed);
System.out.println("After sorting:");
for (i = 0; i < a.length; i++)
    System.out.println(a[i] + ", ");
System.out.println();
\end{verbatim}
Sorting Arrays of Comparable

```java
33 System.out.println("After sorting:");
34 for (i = 0; i < number1; i++)
35 System.out.print(p[i] + " ");
36 System.out.println();
37 }
38 }
```

Sample Dialogue

<table>
<thead>
<tr>
<th>Before Sorting</th>
<th>After sorting:</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.0, 6.0, 5.8, 4.0, 3.0, 2.0, 1.0</td>
<td>1.0, 2.0, 3.0, 4.0, 5.8, 6.0, 7.0, 8.0, 9.0, 10.0</td>
</tr>
<tr>
<td>before sorting:</td>
<td>after sorting:</td>
</tr>
<tr>
<td>dog, cat, cornish game hen</td>
<td>dog, cat, cornish game hen</td>
</tr>
</tbody>
</table>

Defined Constants in Interfaces

- An interface can contain defined constants in addition to or instead of method headings
  - Any variables defined in an interface must be public, static, and final
  - Because this is understood, Java allows these modifiers to be omitted
- Any class that implements the interface has access to these defined constants

Pitfall: Inconsistent Interfaces

- In Java, a class can have only one base class
  - This prevents any inconsistencies arising from different definitions having the same method heading
- In addition, a class may implement any number of interfaces
  - Since interfaces do not have method bodies, the above problem cannot arise
  - However, there are other types of inconsistencies that can arise

Pitfall: Inconsistent Interfaces

- When a class implements two interfaces:
  - One type of inconsistency will occur if the interfaces have constants with the same name, but with different values
  - Another type of inconsistency will occur if the interfaces contain methods with the same name but different return types
- If a class definition implements two inconsistent interfaces, then that is an error, and the class definition is illegal
The **Serializable** Interface

- An extreme but commonly used example of an interface is the **Serializable** interface
  - It has no method headings and no defined constants: It is completely empty
  - It is used merely as a type tag that indicates to the system that it may implement file I/O in a particular way

The **Cloneable** Interface

- The **Cloneable** interface is another unusual example of a Java interface
  - It does not contain method headings or defined constants
  - It is used to indicate how the method `clone` (inherited from the **Object** class) should be used and redefined

The **Cloneable** Interface

- 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 is adequate
  - This is the simple case
- The following is an example of a simple class that has no instance variables of a mutable class type, and no specified base class
  - So the base class is **Object**

Implementation of the Method `clone`: Simple Case

```
public class Tour implements Cloneable {
    public Object clone() {
        try {
            return super.clone(); // Invocation of clone
        } catch (CloneNotSupportedException e) {
            // This should not happen.
            return null; // To keep the compiler happy.
        }
    }
}
```
The Cloneable Interface

- If the data in the object to be cloned includes instance variables whose type is a mutable class, then the simple implementation of `clone` would cause a privacy leak.
- When implementing the `Cloneable` interface for a class like this:
  - First invoke the `clone` method of the base class `Object` (or whatever the base class is)
  - Then reset the values of any new instance variables whose types are mutable class types
  - This is done by making copies of the instance variables by invoking their clone methods.

Implementation of the Method `clone`: Harder Case

```
public class YourCloneableClass implements Cloneable {
    private DataClass someVariable;

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

Simple Uses of Inner Classes

- Inner classes are classes defined within other classes.
  - The class that includes the inner class is called the outer class.
  - There is no particular location where the definition of the inner class (or classes) must be placed within the outer class.
  - Placing it first or last, however, will guarantee that it is easy to find.
Simple Uses of Inner Classes

• An inner class definition is a member of the outer class in the same way that the instance variables and methods of the outer class are members
  – An inner class is local to the outer class definition
  – The name of an inner class may be reused for something else outside the outer class definition
  – If the inner class is private, then the inner class cannot be accessed by name outside the definition of the outer class

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
  – It is legal to invoke a private method of the outer class
• Within the definition of a method of the outer class
  – It is legal to reference a private instance variable of the inner class on an object of the inner class
  – It is legal to invoke a (nonstatic) method of the inner class as long as an object of the inner class is used as a calling object
• Within the definition of the inner or outer classes, the modifiers public and private are equivalent

Simple Uses of Inner Classes

• There are two main advantages to inner classes
  – They can make the outer class more self-contained since they are defined inside a class
  – Both of their methods have access to each other's private methods and instance variables
• Using an inner class as a helping class is one of the most useful applications of inner classes
  – If used as a helping class, an inner class should be marked private

Class with an Inner Class

```java
public class BankAccount
{
    private class Money
    {
        private long dollars;
        private int cents;

        public Money(String stringAmount)
        {
            abortIfNull(stringAmount);
            int length = stringAmount.length();
            dollars = Long.parseLong(
                stringAmount.substring(0, length - 3));
            cents = Integer.parseInt(
                stringAmount.substring(length - 2, length));
        }

        public String getAmount()
        {
            if (cents > 9)
                return (dollars + "." + cents);
            else
                return (dollars + ".0" + cents);
        }
    }

    // Constructor...

    public void deposit(long amount)
    {
        Money m = new Money(String.valueOf(amount));
        // Use m on this BankAccount...
    }
}
```
Class with an Inner Class

The .class File for an Inner Class

- Compiling any class in Java produces a .class file named ClassName.class
- Compiling a class with one (or more) inner classes causes both (or more) classes to be compiled, and produces two (or more).class files
  - Such as ClassName.class and ClassName$InnerClassName.class

Static Inner Classes

- A normal inner class has a connection between its objects and the outer class object that created the inner class object
  - This allows an inner class definition to reference an instance variable, or invoke a method of the outer class
- There are certain situations, however, when an inner class must be static
  - If an object of the inner class is created within a static method of the outer class
  - If the inner class must have static members

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

- Since a static inner class has no connection to an object of the outer class, within an inner class method
  - Instance variables of the outer class cannot be referenced
  - Nonstatic methods of the outer class cannot be invoked
- To invoke a static method or to name a static variable of a static inner class within the outer class, preface each with the name of the inner class and a dot

Public Inner Classes

- If an inner class is marked public, then it can be used outside of the outer class
- In the case of a nonstatic inner class, it must be created using an object of the outer class
  ```java
  BankAccount account = new BankAccount();
  BankAccount.Money amount = account.new Money("41.99");
  ```
  - Note that the prefix `account` must come before `new`
  - The new object `amount` can now invoke methods from the inner class, but only from the inner class

Tip: Referring to a Method of the Outer Class

- If a method is invoked in an inner class
  - If the inner class has no such method, then it is assumed to be an invocation of the method of that name in the outer class
  - If both the inner and outer class have a method with the same name, then it is assumed to be an invocation of the method in the inner class
  - If both the inner and outer class have a method with the same name, and the intent is to invoke the method in the outer class, then the following invocation must be used:
    ```java
    OuterClassName.this.methodName()
    ```
Nesting Inner Classes

• It is legal to nest inner classes within inner classes
  – The rules are the same as before, but the names get longer
  – Given class `A`, which has public inner class `B`, which has public inner class `C`, then the following is valid:
    ```java
    A aObject = new A();
    A.B bObject = aObject.new B();
    A.B.C cObject = bObject.new C();
    ```

Inner Classes and Inheritance

• Given an `OuterClass` that has an `InnerClass`
  – Any `DerivedClass` of `OuterClass` will automatically have `InnerClass` as an inner class
  – In this case, the `DerivedClass` cannot override the `InnerClass`

Anonymous Classes

• If an object is to be created, but there is no need to name the object’s class, then an anonymous class definition can be used
  – The class definition is embedded inside the expression with the `new` operator
• Anonymous classes are sometimes used when they are to be assigned to a variable of another type
  – The other type must be such that an object of the anonymous class is also an object of the other type
  – The other type is usually a Java interface

Anonymous Classes

```java
public class AnonymousClassDemo {
    public static void main(String[] args) {
        NumberCarrier nObject =
            new NumberCarrier() {
                private int number;
                public void setNumber(int value) {
                    number = value;
                }
                public int getNumber() {
                    return number;
                }
            };
    }
}
```
Anonymous Classes

```java
NumberCarrier anotherObject =
    new NumberCarrier()
    {
        private int number;
        public void setNumber(int value)
        {
            number = 2*value;
        }
        public int getNumber()
        {
            return number;
        }
    };

anotherObject.setNumber(42);
showNumber(anotherObject);
showNumber(anotherObject);
System.out.println("End of program.");

public static void showNumber(NumberCarrier n)
{
    System.out.println(n.getNumber());
}
```

**Sample Dialogue**

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

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

This is the file
ANONYMOUS_CARRIERS.java.

Copyright © 2012 Pearson Addison-Wesley. All rights reserved.