Interfaces

CHAPTER 13

Slides prepared by Rose Williams, *Binghamton University*

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

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

Interfaces

- 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

Display 13.1 The Ordered Interface

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

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

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

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

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
     - If more than one interface is implemented, each is listed, separated by commas
  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

Display 13.2 Implementation of an Interface

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

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

Implementation of an Interface

Display 13.2 Implementation of an Interface (continued)

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

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

Display 13.3 An Abstract Class Implementing an Interface

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

19     public abstract boolean follows(Object other);

20 }
```

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

Display 13.4 Extending an Interface

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

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

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

Pitfall: Interface Semantics Are Not Enforced

- When a class implements an interface, the compiler and run-time 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

- Chapter 6 discussed the Selection Sort algorithm, and examined a method for sorting a partially filled array of type `double` into increasing order
- This code could be modified to sort into decreasing order, or to sort integers or strings instead
  - Each of these methods would be essentially the same, but making each modification would be a nuisance
  - The only difference would be the types of values being sorted, and the definition of the ordering
- Using the `Comparable` interface could provide a single sorting method that covers all these cases

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:

```
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"

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

`GeneralizedSelectionSort` class: `sort` Method

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

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

`GeneralizedSelectionSort` class: `sort` Method

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

`GeneralizedSelectionSort` class: `interchange` Method

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

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

}
```

Sorting Arrays of `Comparable`

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

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

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

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

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

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

Sorting Arrays of `Comparable`

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

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

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

Sorting Arrays of `Comparable`

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

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

SAMPLE DIALOGUE

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

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**