

# Chapter 13

## Interfaces and Inner Classes

# 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
20    public abstract boolean follows(Object other);
21 }
```

# 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"

# The Comparable Interface Semantics

- 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

# 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           The classes Double and String do
6  {                                     implement the Comparable interface.
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
13         System.out.println("Before sorting:");
14         int i;
15         for (i = 0; i < d.length; i++)
16             System.out.print(d[i].doubleValue() + ", ");
17         System.out.println();
18
19         GeneralizedSelectionSort.sort(d, d.length);
20
21         System.out.println("After sorting:");
22         for (i = 0; i < d.length; i++)
23             System.out.print(d[i].doubleValue() + ", ");
24         System.out.println();
25     }
26 }
```

# 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

# 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

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

---

```
1  public class YourCloneableClass implements Cloneable
2  {
3      .
4      .
5      .
6      public Object clone()
7      {
8          try
9          {
10             return super.clone(); //Invocation of clone
11                                     //in the base class Object
12         }
13         catch(CloneNotSupportedException e)
14         { //This should not happen.
15             return null; //To keep the compiler happy.
16         }
17     }
18     .
19     .
20     .
21 }
```

*Works correctly if each instance variable is of a primitive type or of an immutable type like `String`.*



# 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

# The Cloneable Interface

- Note that this will work properly only if the **Cloneable** interface is implemented properly for the classes to which the instance variables belong
  - And for the classes to which any of the instance variables of the above classes belong, and so on and so forth
- The following shows an example

# Implementation of the Method `clone`: Harder Case

Display 13.8 Implementation of the Method `clone` (Harder Case)

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

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

*If the clone method return type is DataClass rather than Object, then this type cast is not needed.*

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

# 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

# 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

# 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

# Class with an Inner Class

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

```
1 public class BankAccount
2 {
3     private class Money ← The modifier private in this line should
4     {                               not be changed to public.
5         private long dollars; ← However, the modifiers public and
6         private int cents;           private inside the inner class Money
7                                     can be changed to anything else and it
8         public Money(String stringAmount) would have no effect on the class
9         {                               BankAccount.
10            abortOnNull(stringAmount);
11            int length = stringAmount.length();
12            dollars = Long.parseLong(
13                stringAmount.substring(0, length - 3));
14            cents = Integer.parseInt(
15                stringAmount.substring(length - 2, length));
16        }
17
18        public String getAmount()
19        {
20            if (cents > 9)
21                return (dollars + "." + cents);
22            else
23                return (dollars + ".0" + cents);
24        }
25    }
```



# Class with an Inner Class

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

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

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

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

# Class with an Inner Class

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

---

```
40     private Money balance;
41     public BankAccount()
42     {
43         balance = new Money("0.00");
44     }
45     public String getBalance()
46     {
47         return balance.getAmount();
48     }
49     public void makeDeposit(String depositAmount)
50     {
51         balance.addIn(new Money(depositAmount));
52     }
53     public void closeAccount()
54     {
55         balance.dollars = 0;
56         balance.cents = 0;
57     }
58 }
```

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

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

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

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

# 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

# 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

```
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

# Public Inner Classes

- In the case of a static inner class, the procedure is similar to, but simpler than, that for nonstatic inner classes

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

- Note that all of the following are acceptable

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

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

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

# 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:

*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:

```
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**
- An outer class can be a derived class
- An inner class can be a derived class also

# 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

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

---

```
1 public class AnonymousClassDemo
2 {
3     public static void main(String[] args)
4     {
5         NumberCarrier anObject =
6             new NumberCarrier()
7             {
8                 private int number;
9                 public void setNumber(int value)
10                {
11                    number = value;
12                }
13                public int getNumber()
14                {
15                    return number;
16                }
17            };
```

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

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

```
18     NumberCarrier anotherObject =
19         new NumberCarrier()
20     {
21         private int number;
22         public void setNumber(int value)
23         {
24             number = 2*value;
25         }
26         public int getNumber()
27         {
28             return number;
29         }
30     };

31     anObject.setNumber(42);
32     anotherObject.setNumber(42);
33     showNumber(anObject);
34     showNumber(anotherObject);
35     System.out.println("End of program.");
36 }

37 public static void showNumber(NumberCarrier o)
38 {
39     System.out.println(o.getNumber());
40 }

41 }
```

*This is still the file  
AnonymousClassDemo.java.*

# Anonymous Classes

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

---

### 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  
NumberCarrier.java.*