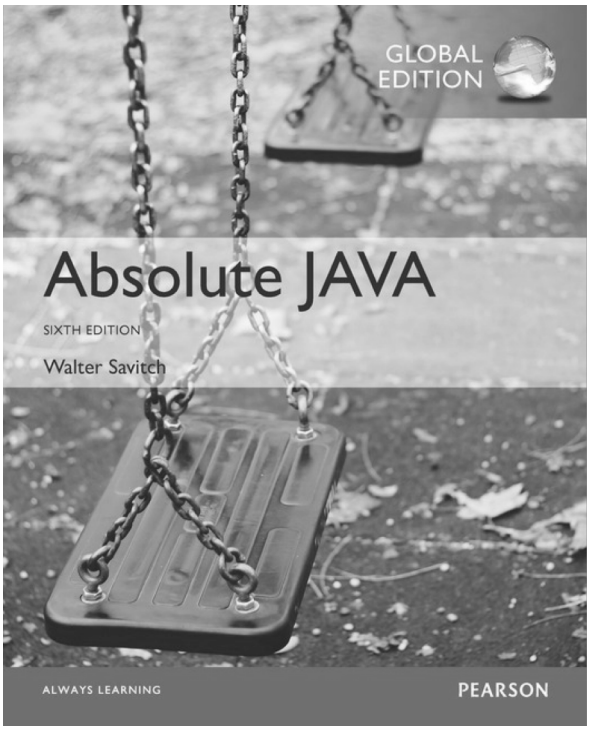

Chapter 13

Interfaces

Slides prepared by Rose Williams, *Binghamton University*

Kenrick Mock, *University of Alaska Anchorage*

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`**

Interfaces

- 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

y 13.1 The Ordered Interface

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

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

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

y 13.2 Implementation of an Interface

```
public class OrderedHourlyEmployee
        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());
        }
    }
```

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

ıy 13.2 Implementation of an Interface (continued)

```
    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*

isplay 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

ıy 13.4 Extending an Interface

```
public interface ShowablyOrdered extends Ordered
{
    /**
      Outputs an object of the class that precedes the calling object.
    */
    public void showOneWhoPrecedes();
}
```

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

Pitfall: Interface Semantics Are Not Enforced

- 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

- Imagine a method for sorting a partially filled array of type `double` into increasing order
- This method 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 **`Comparable`** Interface Semantics

- 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 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

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

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

.5 Sorting Method for Array of Comparable *(Part 1 of 2)* (continued)

```
/**
 Returns the index of the smallest value among
 a[startIndex], a[startIndex+1], ... a[numberUsed - 1]
*/
private static int indexOfSmallest(int startIndex,
                                   Comparable[] a, int numberUsed)
{
    Comparable min = a[startIndex];
    int indexOfMin = startIndex;
    int index;
    for (index = startIndex + 1; index < numberUsed; index++)
        if (a[index].compareTo(min) < 0)//if a[index] is less than min
        {
            min = a[index];
            indexOfMin = index;
            //min is smallest of a[startIndex] through a[index]
        }
    return indexOfMin;
}
```

`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]
    }

}
```



```
/**
 Demonstrates sorting arrays for classes that
 implement the Comparable interface.
*/
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(d.length - i);

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

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

Sorting Arrays of **Comparable**

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

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

System.out.println("Before sorting:");
for (i = 0; i < numberUsed; i++)
    System.out.print(a[i] + ", ");
System.out.println();

GeneralizedSelectionSort.sort(a, numberUsed);
```


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 constants 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`**

ay 13.7 **Implementation of the Method clone (Simple Case)**

```
public class YourCloneableClass implements Cloneable
{
        .                          Works correctly if each instance variable is of a
        .                          primitive type or of an immutable type like String.
        .
    public Object clone()
    {
       try
       {
          return super.clone();//Invocation of clone
                               //in the base class Object
       }
       catch(CloneNotSupportedException e)
       {//This should not happen.
          return null; //To keep 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

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

isplay 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
0          {
1              YourCloneableClass2 copy =
2                                  (YourCloneableClass2)super.clone();
3              copy.someVariable = (DataClass)someVariable.clone();
4              return copy;
5          }
6          catch(CloneNotSupportedException e)
7          {//This should not happen.
8              return null; //To keep the compiler happy.
9          }
0      }
1          .
2          .
3          .
```

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` *rathe than* `Object`*, then this type cast is not needed.*