Lecture - Interfaces

Interfaces is not a class
It is a type of java.

Interfaces

Interface define the characteristic or properties of a class.
in term of method.

An *interface* is essentially a set of methods that a class promises to **implement**. Interfaces are used in Java to achieve and improve the Abstraction.

It 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 (i.e., defines the required methods).

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 serves a function similar to a base class, though it is not a base class.

Some languages (like C++) 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.

Example: Given a base class `Vehicle`, we may want to make classes `AirPlane`, `Boat` and `Car`. But what if a vehicle is both an airplane and a boat? Instead, we could define *interfaces* `Flyable` and `Floatable`, and create a class like `Seaplane` which extends `Vehicle`, but also implements the `Flyable` and `Floatable` interfaces.

Public methods

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 (just as a derived class cannot give a more restrictive permission to any overridden method).

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 an object of any class that implements the interface.

Example: The `Ordered` interface

```
public interface Ordered {
    public boolean precedes (Object other);     // don't forget the semicolon

    /**
     For objects o1 and o2 of the class, we should have
     o1.follows(o2) == o2.precedes(o1)
     However, neither the compiler nor run-time system will ensure this.
     It is only advisory to the programmer implementing the interface.
    */
    public boolean follows (Object other);
}
```

interface can only have public methods.

Implementing an interface

To implement an interface, a concrete class must do two things:

- It must include the phrase

```
    implements InterfaceName
```

at the start of the class definition. If more than one interface is implemented, each is listed, separated by commas.

- The class must implement **all** the method headings listed in the definition(s) of the interfaces(s).

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

can have many interface
but only one base class (inherit)

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

   public static void main(String[] args) {
       // Fill me in
   }
}

interface Ordered {
    public boolean precedes (Object other);     // don't forget the semicolon
```

not body in interface.

```
    /**
     For objects o1 and o2 of the class, we should have
     o1.follows(o2) == o2.precedes(o1)
    */
    public boolean follows (Object other);
}

class HourlyEmployee {
    public double getPay () {
        return 0;
    }
}
```

Exercise: Read and understand the above code. Write a `main` method to create two `HourlyEmployee` objects and compare them.

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 have definitions for all the method headings given in the abstract class *and the interface*. Notice in the following example that the concrete class doesn't need to *redefine* the method precedes, because it is already defined in the abstract class.

```
public class MyConcreteClass extends MyAbstractClass {
    public boolean follows (Object other) {
        return true;        // Write the correct function here
    }

    public static void main(String[] args) {

    }
}

abstract class MyAbstractClass implements Ordered {
    int number;
    char grade;

    public boolean precedes (Object other) {
        if (other == null)
            return false;
        else if (!(other instanceof MyAbstractClass))
            return false;
        else {
            MyAbstractClass otherMyAbstractClass
                                    = (MyAbstractClass)other;
            return (number < otherMyAbstractClass.number);
        }
    }

    public abstract boolean follows (Object other);
}
```

(if we don't write this method, it still works)

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

```
public interface ShowablyOrdered extends Ordered {
    /**
     Outputs an object of the class tat precedes the calling object.
     (Neither the compiler nor the run-time system will ensure this is satisfied)
    */
    public void showOneWhoPrecedes();
}
```

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.

The Comparable interface

The standard way of testing if an object is "less than" an object of the same time is 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 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. (We will cover exceptions next week).

Semantics

The "semantics" of code refers to what the code "means". It is contrasted with "syntax", which refers to what code is "valid".

The semantics of `compareTo` are stated above, in terms of "comes before" and "comes after". But what do these actually mean?

Almost any reasonable notion of "comes before" is acceptable. For example:

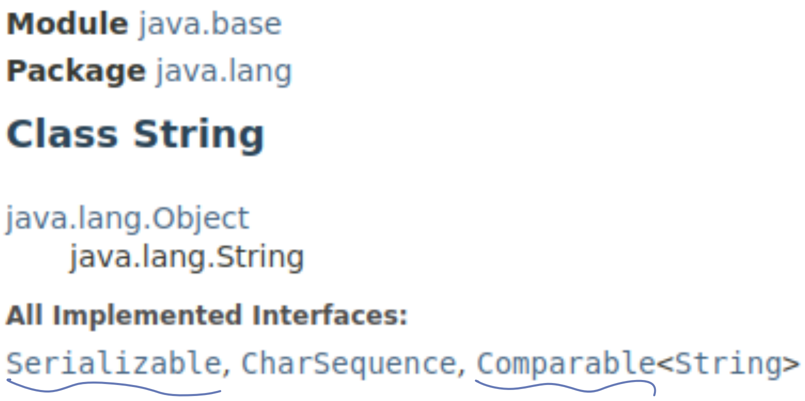
- the standard less-than relations on numbers (used by types Double, Integer, etc.)
- lexicographic ordering on strings (used by String; as governed by the locale)
- case-sensitive ordering of strings
- case-insensitive ordering of strings
- lexicographic ordering of tuples ((A,B) < (C,D) if A<C or (A == C and B < D))
- ordering Person by age, or by date of birth, or by surname, or by given name

The relationship "comes after" is just the reverse of "comes before".

Example application: sorting (little confuse)

The following example reworks the `SelectionSort` class from Programming with arrays in week 5 (Chapter 6 of the text book).

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.



Module java.base
Package java.lang

Class String

java.lang.Object
      java.lang.String

All Implemented Interfaces:

Serializable, CharSequence, Comparable<String>

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.

```
/**
 Demonstrates sorting arrays for
 classes implement the Comparable interface
*/
public class ComparableDemo {
    public static void main(String[] args) {
        // Example 1: numbers
        Double[] d = new Double[10];        // Double implements Comparable

        // initialized
        for (int i = 0; i < d.length; i++)
            d[i] = d.length - (double)i;

        System.out.println("Before sorting:");
        for (double v : d) {
            System.out.print(v + ", ");
        }
        System.out.println();

        // Perform sort
        GeneralizedSelectionSort.sort(d, d.length);

        // Check results
        System.out.println("After sorting:");
        for (double v : d)
            System.out.print(v + ", ");
        System.out.println();

        // Example 2: Strings
        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 (int i = 0; i < numberUsed; i++) {
            System.out.print(a[i] + ", ");
        }
        System.out.println();

        // Perform sort
        GeneralizedSelectionSort.sort(a, numberUsed);

        // Check results
        System.out.println("After sorting:");
        for (int i = 0; i < numberUsed; i++) {
            System.out.print(a[i] + ", ");
        }
        System.out.println();
    }
}

class GeneralizedSelectionSort {
    /**
     Precondition:  numberUsed < a.length;
                The first numberUsed indexed valuables have values (not null)
     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);
            // Invariant:
            // 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.
        }
    }

    /**
     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;
                // Invariant:
                // min is smallest of a[startIndex], ..., a[index]
            }
        return indexOfMin;
    }

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

Exercise: Identify where the `Comparable` interface is important -- either used, or an object implementing that interface is created, if the program actually uses the interface on that object.

Excercise: Define a `memorySize` class that stores a string of the form "<digits><multiplier>", where <digits> is a string consisting of one or more digits in the range 0-9, and <multiplier> is empty, 'k', 'M', 'G', 'T', 'P', 'ki', 'Mi', 'Gi', 'Ti' or 'Pi'. A multiplier of 'k' multiplies the number represented by <digits> by 1000. A multiplier of 'ki' multiplies by 1024. A multiplier of 'M' multiplies by 10^6. A multiplier of 'Mi' multiplies by (1024)^2 etc.. Define `compareTo` correctly so that, for example, 1 < 1020k < 1000ki.

Use your class to sort   10, 2, 1005, 9k, 8ki, 7M, 6G, 5Mi, 40G.

Advanced: Modify `memorySize` to store not only the string but also the integer represented by the string, so that `compareTo` can simply compare two integers. Time the two versions and compare their run times.

Exercise: Identify where automatic boxing is used. Identify where automatic unboxing is used. Where the for(...;...;...) form of loops is used, explain why a for-each is not suitable.

Example application: searching

It is common to want to find an item in a sorted list. The most efficient way to do this is a binary search: Check if the item in the middle of the list is greater than or less than the item we are searching for. If it is less, then we continue searching the right half of the list. If it is more, we search the left half. Since this only requires greater-that / less-than / equal to comparison, we can write generic code to search for an item of any class that implements Comparable.

This ability to write generic code is important for writing complex software without having to have very repetitive code, and is one of the strengths of object oriented programming.

(short) Defined constants in interfaces

Although an interface cannot contain any member variables, it **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.

Inconsistent interfaces

diamond problem.

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. 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 and signature but different *return types*

If a class definition implements two inconsistent interfaces, then that is an error, and the class definition is **illegal**.

Another more subtle problem is that both interfaces can have the same method heading but with different intended semantics. (little confused)

- For example, a method `same` can be intended in one interface to return true if two objects have equal value (like `equals`) and in the other interface be intended to return true if two references refer to the same object (like `==`). A class implementing `boolean same (Object o)` will satisfy the required syntax of both interfaces (i.e., it will be legal), but will probably cause unintended behaviour.
- Alternatively, an interface could capture the idea of partial ordering. For example, (a,b) < (c,d) if a<c and b<d, (a,b) > (c,d) if a>c and b>d, but they are "incomparable" otherwise. If a programmer carelessly required a method `compareTo` of this class to return 0 for incomparable pairs, then a class implementing both this interface and `Comparable` would run into trouble in a binary search. (What trouble?) For this reason, it is good practice not to reuse the names of methods of standard java interfaces. If in doubt, search the web for the name you are considering and see if you find it in an existing interface.