Object Oriented Programming

Java

Part 7: Interfaces

Interfaces – revision (1)

- An **interface** is a collection of operations (without implementation) that are used to specify a service of a class:
  - Interfaces do not contain attributes, except for constants.
  - An interface may be realized (or implemented) by a **concrete class**. In this **implementation** or **realization**:
    - The attributes needed to the correct implementation of the interface are determined.
    - The code of the methods made specified by the interface is provided.

Interfaces – revision (2)

- An interface may inherit the definitions of another interface.
  - Interfaces may use polymorphism.
  - If a class realizes more than one interface, and different interfaces have methods with the same signature, the class should provide only one implementation of those methods.
- An interface cannot be instantiated.

Interfaces (1)

- Both classes and interfaces define **types** (the fundamental unit of OO programming).
- In Java, a class can only extend another class, but it might implement more than one interface.
- For a given class the extended class and implemented interfaces are its **supertypes**. The class itself is the **subtype**.
- A reference to an object of the subtype can always be used when a reference to an object of the supertypes (classes or interfaces) is required.

Interfaces (2)

- Java provides a set of interfaces, being the most used:
  - `Comparable`: objects of this type have an associated order making possible to compare them.
  - `Iterable`: objects of this type provide an iterator and so they can be used in a enhanced for-each loop.
    - `Collection`: objects of this type store other objects.
      - `Set`, `List`, `Queue`, ...

Interfaces (3)

Syntax

Modifier* interface Ident [extends IdentI [, IdentI]*] {
[ModifierC* Type IdC = expression;] *
[ModifierM* Type IdM ([TypeP IdP [, TypeP IdP]*);]*
}

- Modifier: modifier (visibility, among others)
- Ident: interface name
- **extends** IdentI: interface specialization

Interfaces (4)

- **Interface modifiers**:
  - **public**: publicly accessible interface.
  - **abstract**: interfaces cannot be instantiated.
- When the **public** modifier is omitted the interface is only accessible in the package where it is defined.

Interfaces (5)

- All interfaces are implicitly **abstract**. By convention, the **abstract** modifier is omitted.
- All interface members are implicitly **public**. By convention, the **public** modifier is omitted.
- All interface constants are implicitly **public static final**. By convention, the modifiers are omitted.
- All interface methods are implicitly **public abstract**. By convention, the modifiers are omitted.
- No other qualifier is allowed to constants and methods of an interface.

Interfaces (6)

```
public interface Queue {
     //methods
     boolean empty();
     Object top ();
     boolean add(Object o);
     void remove();
}
```

Inheritance of interfaces (1)

- An interface may extend more than one interface.
- The extended interfaces are denominated **superinterfaces**, whereas the new interface is named **subinterface**.
- A subinterface inherits all constants declared in its superinterfaces.
- If a subinterface declares a constant with the same name as one inherited from its superinterfaces (independently of the type), the subinterface constant **hides** the inherited constant.
- In the subinterface the inherited constant is accessed only by its qualified name (`superinterface.constant`).

Inheritance of interfaces (2)

```
interface X {
     int val = 1;
     String strx = "X";
}
```

```
interface Y extends X {
     int val = 2;
     int sum = val + X.val;
     String stry = "Y extends" + strx;
}
```

Inheritance of interfaces (3)

- If an interface inherits two or more constants with the same name, any non-qualified use of that constant is ambiguous and it results in a compile-time error.

```
interface A {
    String str = "A";
}
```

```
interface B extends A {
    String str = "B";
}
```

```
interface C extends A {
    String str = "C";
}
```

```
interface D extends B, C {
    String d = str;
}
```

```
interface D extends B, C {
    String d = A.str+B.str+C.str;
}
```

Compile-time error: which `str`?

Inheritance of interfaces (4)

- A subinterface all methods declared in its superinterfaces.
- If a subinterface declares a method with the same signature, up to a covariant return, as one or more methods inherited from the superinterfaces, the method in the subinterface is a **redefinition** of the inherited methods.
- If a subinterface inherits more than one method with the same signature, up to a covariant return, the subinterface contains only one method – the method that returns the common subtype (or one bellow in the hierarchy).

Inheritance of interfaces (5)

- If a subinterface method differs only in the return type of an inherited method, or two inherited methods differ only in the return type, and these returns are not covariant, there is a compilation error.
- If a subinterface method has the same name but different parameters of an inherited method, the subinterface method is an **overload** of the inherited method.

Inheritance of interfaces (6)

```
interface X {
    void xpto();
    Number foo1();
    Number foo2();
}
```

```
interface Y {
    Object foo1();
    Object foo2();
}
```

```
interface Z extends X, Y {
    void xpto(String s);
    Integer foo1();
}
```

- Methods of the interface `Z`:
  - `public void xpto()`
  - `public void xpto(String)`
  - `public Integer foo1()`
  - `public Number foo2()`

Implementation of interfaces (1)

- A class identifies the interfaces that implement, listing them after the keyword `implements`.

Syntax (revision)

Modifier* class Ident

[extends IdentC] [implements IdentI [,IdentI]*] {

[Fields | Methods]*

}

Implementation of interfaces (2)

- The interfaces that a class implements are denominated **superinterfaces** of the class.
- The class should provide an implementation for all methods defined in the superinterfaces, otherwise the class must be declared as **abstract**.

Implementation of interfaces (3)

- When a class implements an interface, the class can access the constants defined in the interface as if they were declared in the class.
- A class that implements more than one interface, or extends a class and implements one or more interfaces, suffers from the same problems of hidden constants and ambiguity that an interface which extends an interface (see slides 10, 11 and 12).

Implementation of interfaces (4)

```
interface X {
     int val = 1;
     String strx = "X";
}
```

```
interface Y extends X {
     int val = 2;
     int sum = val + X.val;
     String stry = "Y extends " + strx;
}
```

```
class Z implements Y {
     int val = 3;
}
```

```
Z z = new Z();
System.out.println(
     "z.val=" + z.val +
     " ((Y)z).val=" + ((Y)z).val + /* ou Y.val */
     " ((X)z).val=" + ((X)z).val)  /* ou X.val */;
System.out.println("z.strx=" + z.strx + " z.stry=" + z.stry;
```

In the terminal is printed

```
z.val=3 ((Y)z).val=2 ((X)z).val=1
strx=X stry=Y extends X
```

Implementation of interfaces (5)

```
interface A {
    String str = "A";
}
```

```
interface B extends A {
    String str = "B";
}
```

```
interface C extends A {
    String str = "C";
}
```

```
class D implements B, C {
    String d = str;
}
```

```
class D implements B, C {
    String d = A.str+B.str+C.str;
}
```

Compile-time error: which `str`?

Implementation of interfaces (6)

- If a class implements multiple interfaces with more than one method having the same signature class contains only one such method.
- If a class implements multiple interfaces with more than one method with the same signature, up to a covariant return, the implementation must define the method that returns the common subtype (otherwise results in a compile error).
- If a class implements multiple interfaces with more than a method that differs only in the return type, and these returns are not covariant, there is a compilation error.

Implementation of interfaces (7)

```
interface X {
    void xpto();
    Number foo1();
    Number foo2();
}
```

```
interface Y {
    Object foo1();
    Object foo2();
}
```

```
interface Z extends X, Y {
    void xpto(String s);
    Integer foo1();
}
```

```
class ClassZ implements Z {
    public void xpto() {...}
    public void xpto(String s) {...}
    public Integer foo1() {...}
    public Number foo2() {...}
}
```

It is important to identify the methods to implement in an interface: if `ClassZ` does not provide an implementation of all methods defined in the superinterfaces it must be declared as **abstract** (see slide 17).

Implementation of interfaces (8)

- An implementation of the `Queue` interface may be done in two different ways:
  - Based on an array: `ArrayQueue`
  - Based on a linked list: `LinkedListQueue`
- The `Queue` interface corresponds to the **abstract data type**, whereas both `ArrayQueue` and `LinkedListQueue` correspond to the **data type** implemented in two different ways.

```
public interface Queue{
     //methods
     boolean empty();
     Object top();
     boolean add(Object obj);
     void remove();
}
```

Implementation of interfaces (9)

```
public class ArrayQueue implements Queue {

    private final int MAX;
    private Object queue[];
    private int freePos; // first free position

    public ArrayQueue(int max) {
        MAX = max;
        queue = new Object[MAX];
        freePos = 0;
    }
    public boolean empty() {
        return freePos==0;
    }
    public Object top() {
        return freePos>0 ? queue[freePos-1] : null;
    }
```

Implementation of interfaces (10)

```
    //continued from previous slide

    public boolean add(Object obj){
        if (freePos<MAX-1){
            queue[freePos++] = obj;
            return true;
        }
        return false;
    }
    public void remove() {
        if (freePos>0)
            queue[--freePos] = null;
    }
    public int nbMaxElements() {
        return MAX;
    }
}
```

Implementation of interfaces (11)

```
public class LinkedListQueue implements Queue {

    private QueueElement base;
    private int nbElements;

    public LinkedListQueue(){
        base = null;
        nbElements = 0;
    }
    public boolean empty() {
        return nbElements==0;
    }
    public Object top() {
        return base!=null ? base.element : null;
    }
```

Implementation of interfaces (12)

```
    //continued from previous slide

    public boolean add(Object obj){
        base = new QueueElement(obj,base);
        nbElements++;
        return true;
    }
    public void remove () {
        if (base!=null) {
            base = base.next;
            nbElements--;
        }
    }
    public int nbElements() {
        return nbElements;
    }
}
```

Implementation of interfaces (13)

```
public class QueueElement{

     Object element;
     QueueElement next;

     public QueueElement(Object elem, QueueElement n){
          element = elem;
          next = n;
     }
}
```

Implementation of interfaces (14)

- As interfaces define a type, it is possible to declare variables/ fields of that type:

```
Queue p = new ArrayQueue(100);
```

- However, references to an interface type, can only be used to access the members of the interface:

```
p.add(new Integer(100));
p.add(new Character('a'));
p.remove();
```

```
int max = p.nbMaxElements(); //INVALID!!!
```

- A cast can be used:

```
int max = ((ArrayQueue)p).nbMaxElements();
```

Implementation of interfaces (15)

- It is possible to invoke any method from `Object` with a reference to an interface:

```
String s = p.toString();
```

Implementation of interfaces (16)

- The programmer may instantiate any queue:

```
Queue p1 = new LinkedListQueue();
Queue p2 = new LinkedListQueue();
```

```
p1.add(new Integer(5));
p2.add(new Character('a'));
```

- If later the programmer decide to use instead an `ArrayQueue` only the instantiation needs to be updated, and not the declared type of the reference (of type `Queue`):

```
Queue p1 = new ArrayQueue(20);
Queue p2 = new ArrayQueue(100);
```

```
p1.add(new Integer(5));
p2.add(new Character('a'));
```