Interfaces

P. Shamoi, 2017

"Walking on water and developing software from a specification are easy if both are frozen."

Edward Berard

Concept

- An **`interface`** is a way to describe what classes should do, without specifying how they should do it. It's not a class but a set of requirements for classes that want to conform to the interface

E.g.

```
public interface Comparable
{
        int compareTo(Object otherObject);
}
```

this requires that any class implementing the `Comparable` interface contains a `compareTo` method, and this method must take an `Object` parameter and return an integer

Interface declarations

- The declaration consists of a keyword `interface`, its name, and the members
- Similar to classes, interfaces can have inside:
  - constants (fields)
  - methods

Interface member – constants

- An interface can define named constants, which are `public`, `static` and `final` (these modifiers are omitted by convention) automatically. Interfaces never contain instant fields.
- All the named constants MUST be initialized

An example interface

```
Interface Verbose {
        int SILENT = 0;
        int TERSE = 1;
        int NORMAL = 2;
        int VERBOSE = 3;

        void setVerbosity (int level);
        int getVerbosity();
}
```

```
interface Educated {
        int PUPIL = 0;
        int BACHELOR = 1;
        int MASTER = 2;
        int PHD = 3;
        void setEducationLevel (int level);
        int getEducationLevel();
}
```

Interface member – methods

- They are implicitly **`abstract`** (omitted by convention). So every method declaration consists of the method header and a semicolon.
- They are implicitly **`public`** (omitted by convention). No other types of access modifiers are allowed.
- They can't be `final`, nor `static`

Modifiers of interfaces itself

- An interface can have different modifiers as follows
  - `public/package(default)`
  - `abstract`
    - all interfaces are implicitly `abstract`
    - omitted by convention

Implementing interfaces in a class

- Two steps to make a class implement an interface

  1. declare that the class intends to implement the given interface by using the **implements** keyword

```
class Employee implements Comparable { . . . }
```

  2. supply definitions for all methods in the interface

```
public int compareTo(Object otherObject) {
    Employee other = (Employee) otherObject;
    if (salary < other.salary) return -1;
    if (salary > other.salary) return 1;
    return 0; }
```

- If a class leaves any method of the interface undefined, the class becomes `abstract` class and must be declared `abstract`

- A single class can implement multiple interfaces. Just separate the interface names by comma

```
class Employee implements Comparable, Cloneable {. . .}
```

Instantiation properties of interfaces

- Interfaces are not classes. You can never use the `new` operator to instantiate an interface.

```
public interface Comparable {
    . . . }
Comparable x = new Comparable( );
```

- You can still declare interface variables

```
Comparable x;
```

but they must refer to an object of a class that implements the interface

```
class Employee implements Comparable {
            . . .
}
x = new Employee( );
```

Extending interfaces

- Interfaces support multiple inheritance – an interface can extend more than one interface
- Superinterfaces and subinterfaces

Example

```
public interface SerializableRunnable extends
java.io.Serializable, Runnable {

        . . .

}
```

Extending interfaces – about constants (1)

- An extended interface inherits all the constants from its superinterfaces
- Take care when the subinterface inherits more than one constants with the same name, or the subinterface and superinterface contain constants with the same name — always use sufficient enough information to refer to the target constants

Tedious Details (1)

- When an interface inherits two or more constants with the same name
  - In the subinterface, explicitly use the superinterface name to refer to the constant of that superinterface

E.g.

```
interface A {
    int val = 1;
}
interface B {
    int val = 2;
}

interface C extends A, B {
    int val = A.val;
}
```

Tedious Details (2)

- If a superinterface and a subinterface contain two constants with the same name, then the one belonging to the superinterface is **hidden**

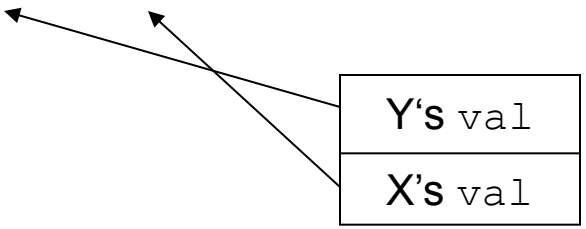
  In the subinterface:

  - access the subinterface-version constants by directly using its name
  - access the superinterface-version constants by using the superinterface name followed by a dot and then the constant name

E.g

```
interface X {
    int val = 1; }

interface Y extends X{
    int val = 2;
    int sum = val + X.val; }
```



Tedious Details (3)

- When a superinterface and a subinterface contain two constants with the same name, and a class implements the subinterface
  - the class inherits the subinterface-version constants as its static fields. Their access follow the rule of class's static fields access.

E.g

```
class Z implements Y { }

//inside the class
System.out.println("Z.val:"+val);    //Z.val = 2

//outside the class
System.out.println("Z.val:"+Z.val); //Z.val = 2
```

  - object reference can be used to access the constants
    - subinterface-version constants are accessed by using the object reference followed by a dot followed by the constant name
    - superinterface-version constants are accessed by explicit casting

E.g.

```
Z v = new Z( );
System.out.print( "v.val = " + v.val
                  +", ((Y)v).val = " + ((Y)v).val
                  +", ((X)v).val = " + ((X)v).val );
```

output: v.val = 2, ((Y)v).val = 2, ((X)v).val = 1

Extending interfaces – about methods

- If a declared method in a subinterface has the same signature as an inherited method and the same return type, then the new declaration *overrides* the inherited method in its superinterface.

- If the only difference is in the return type, then there will be a compile-time error

- An interface can inherit more than one methods with the same signature and return type. A class can implement different interfaces containing methods with the same signature and return type.

- Methods with same name but different parameter lists are **`overloaded`**

Why using interfaces?

The usefulness of interfaces goes far beyond simply publishing protocols for other programmers.

Any function can have parameters that are of interface type.

Any object of a class that implements the interface may be passed as an argument.

```
public interface Vegetarian{}
public class Animal{}
public class Deer extends Animal implements Vegetarian{}
```

```
class Employee extends Person implements Vegetarian{}

 public void doSmth(Vegetarian v){}
```

Marker interfaces and object cloning

- *A marker interface* has neither methods nor constants, its only purpose is to allow the use of **`instanceof`** in a type inquiry. **`Cloneable`** interface is such an example.
- *Object clone*: a **clone** method returns a new object whose initial state is a copy of the current state of the object on which `clone` was invoked. Subsequent changes to the new clone object should not affect the state of the original object.
- Three factors in writing a `clone` method
  - The empty Cloneable interface. You must implement it to provide a clone method that can be used to clone an object
  - The clone method implemented by the Object class performs a simple clone by copying all fields of the original object to the new object
  - The CloneNotSupportedException, which can be used to signal that a class's clone method shouldn't have been invoked

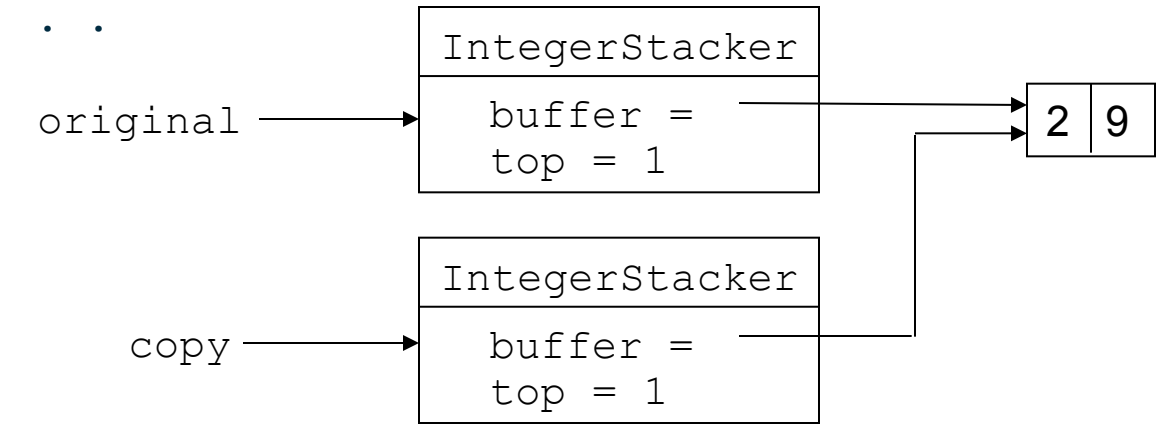
Object cloning (1)

- The **`Object`** class provides a method named **`clone`**, which performs a simple clone by *copying all fields of the original object to the new object*. It works for many classes but may need to be overridden for special purpose.

- **Shallow** versus **deep** cloning

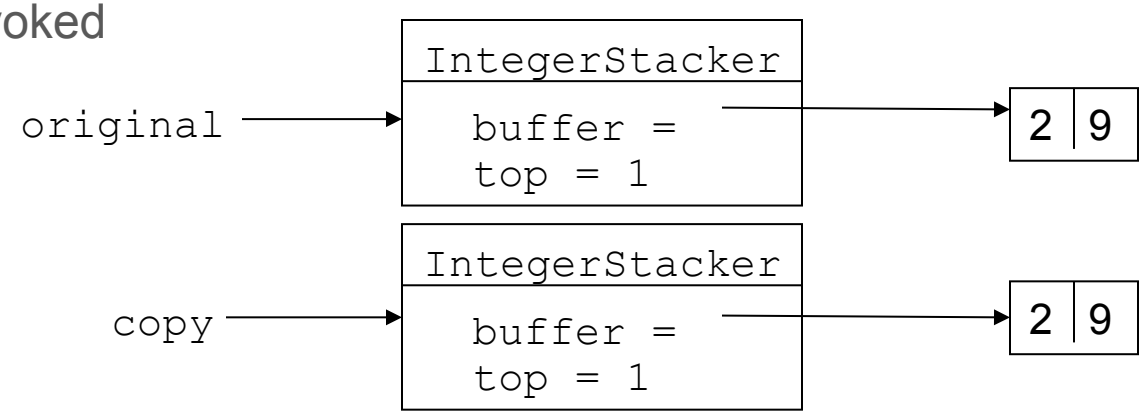
  1) **Shallow cloning:** a simple field by field copy. *This might be wrong if it duplicates a reference to an object that shouldn't be shared.*

```
public class IntegerStack implements Cloneable {
   private int[] buffer; // a stacker of integers
   private int top;      // largest index in the stacker
                         // (starting from 0)
   . . .
}
```



Object cloning (2)

2) **Deep cloning**: cloning all of the objects from the object on which clone is invoked



- In order to be **cloneable**, the class must:
  1. Implement the **`Cloneable`** interface
     - **`Cloneable`** interface has neither methods nor constants, but marks a class as partaking in the cloning mechanism
  2. Redefine the **`clone`** method with the **`public`** access modifier

- If you decide that a class just needs shallow cloning, you still need to implement the `Cloneable` interface, redefine `clone` to be `public`, and call `super.clone()`

The Cloneable Interface

Again, marker Interface is an empty interface.

A marker interface does not contain constants or methods, but it has a special meaning to the Java system. The Java system requires a class to implement the Cloneable interface to become cloneable.

```
public interface Cloneable
{ // no code there

}
```

Example of Creating an Interface

```
// This interface is defined in
// java.lang package
public interface Comparable
{
  public int compareTo(Object o);
}
```

Generic max Method

```
public class Max
{
  // Return the maximum between two objects
 public static Comparable max(Comparable o1, Comparable o2)
  {
    if (o1.compareTo(o2) > 0)
      return o1;
    else
      return o2;
  }

 }
```

Interfaces and abstract classes

- Why bother introducing two concepts: abstract class and interface?

```
abstract class Comparable  {
   public abstract int compareTo (Object otherObject);
}
class Employee extends Comparable  {
    pulibc int compareTo(Object otherObject) { . . . }
}
```

```
public interface Comparable {
    int compareTo (Object otherObject)
}
class Employee implements Comparable  {
    public int compareTo (Object otherObject) { . . . }
}
```

- A class can only extend a single abstract class, but it can implement as many interfaces as it wants
- An abstract class can have a partial implementation, fields and methods, while interfaces are limited to public constants and public methods with no implementation

Interfaces and abstract classes

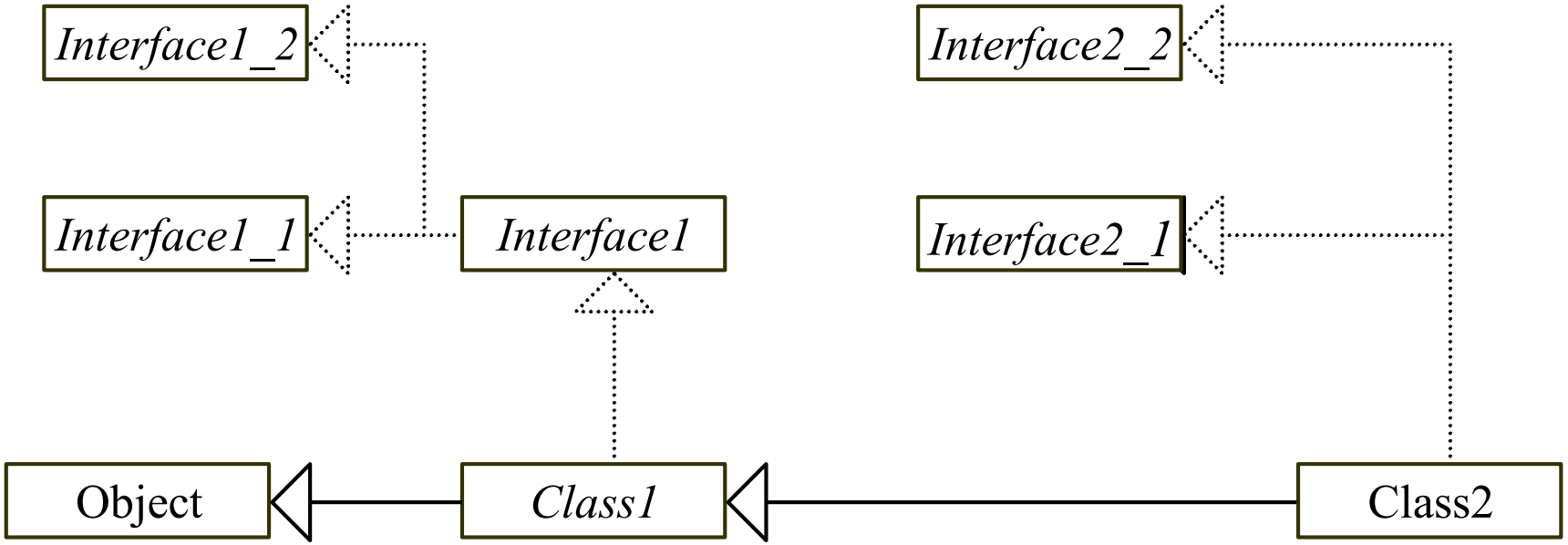
Again, in an interface, the data **must** be constants an abstract class can have **all types** of data.

Each method in an interface has **only a signature** without implementation, but an abstract class can have **concrete methods**.

An abstract class **must** contain at least one abstract method.

Note: since all the methods defined in an interface are abstract methods, Java does not require you to put the abstract modifier in the methods in an interface, but you must put the abstract modifier before an abstract method in an abstract class.

Interfaces vs. Abstract Classes (cont.)



From the figure above, think of the following:

Can a class extend multiple classes?

Can a class implement multiple interfaces?

Can an interface extend multiple interfaces?