Programmation Objet
Java – Interfaces

Michail Lampis
michail.lampis@dauphine.fr

Rappel : Classe Set

- Classe abstraite

```
abstract class Set {
        abstract public Object [] toArray();
        abstract public String toString();
        abstract public Set Copy() ;
        abstract public void add(Object o);
        abstract public boolean isMember(Object o);
        public Set Union(Set s){..};
        abstract public boolean isEmpty();
}
```

Rappel : Classe Stack

```
public abstract class Stack {
        abstract public void push(Object o);
        abstract public Object pop();
        abstract public String toString();
        abstract public Object top() ;
}
```

Héritage Multiple

- On peut envisager de créer un ADT qui implémente à la fois les fonctionnalités de **Set** et **Stack**.
  - Motivation : Une pile qui nous permet de faire de testes d'appartenance.

    SetStack s = new … ;

    s.push(5) ;

    s.push(7) ;

    s.isMember(5) == true
- Quelle est la définition de **Union** dans ce cas ?
  - Plusieurs possibilités...

Héritage Multiple

- Une tentative raisonnable :

  class mySetStack extends Set,Stack {

  … }

- → Erreur de compilation !
  - Chaque classe ne peut étendre qu'une autre classe.
  - Pas d'héritage multiple en Java
    - (Par contre, permis en C++…)
- Solution : Interfaces

Interface

- Une interface est une définition abstraite d'une fonctionnalité
  - (Normalement) on définit que des méthodes (pas des champs)

```
interface interA{
        /* abstract public */ void f();
        int g(double x);
}
class A implements interA{
        void f(){  System.out.println("A.f");    }
        int g(){  System.out.println("A.g");  return 0;  }
}
```

Interface

- On suit les règles habituelles de l'héritage.
  - Toute les méthodes de l'interface sont automatiquement **abstract public**
  - → Une sous-classe qui implémente l'interface doit les redéfinir (override)

```
interface interA{
        /* abstract public */ void f();
        int g(double x);
}
class A implements interA{
        void f(){  System.out.println("A.f");    }
        int g(){  System.out.println("A.g");  return 0;  }
}
```

Interface

- On suit les règles habituelles de l'héritage.
  - Toute les méthodes de l'interface sont automatiquement **abstract public**
  - → Une sous-classe qui implémente l'interface doit les redéfinir (override)

```
interface interA{
        /* abstract public */ void f();
        int g(double x);
}
class A implements interA{
public  void f(){  System.out.println("A.f");    }
        int g(){  System.out.println("A.g");  return 0;  }
}
```

Interfaces - Polymorphisme

- On peut utiliser une interface comme un type
  - Interface ≈ classe abstraite
  - Différences :
    - Interface ne contient pas de champs
      - (sauf **public static final)**
    - Méthodes d'une interface sont public, habituellement abstract
      - Exception à voir plus tard (**default**)
    - Une classe peut implémenter plusieurs interfaces, mais étendre une seule autre classe.

Interfaces - Polymorphisme

```
interface interA{ void f(); }
class B{ void h(){ System.out.println("B.h"); }}
class A extends B implements interA{
        public void f(){ System.out.println("A.f"); }
        public static void main(String []args){
                A a = new A();
                a.f();
                a.h();
        }
}
```

Interfaces - Polymorphisme

```
interface interA{ void f(); }
class B{ void h(){ System.out.println("B.h"); }}
class A extends B implements interA{
        public void f(){ System.out.println("A.f"); }
        public static void main(String []args){
                interA a = new A();
                a.f();
                a.h(); //N'existe pas !?
        }
}
```

Interfaces - Polymorphisme

```
interface interA{ void f(); }
class B{ void h(){ System.out.println("B.h"); }}
class A extends B implements interA{
        public void f(){ System.out.println("A.f"); }
        public static void main(String []args){
                B a = new A();
                a.f();    //N'existe pas !?
                a.h();
        }
}
```

Interfaces

- Polymorphisme → on peut utiliser des variables dont le type est une interface
  - Signification → on attend à un objet dont le type réel sera une classe qui implémente l'interface
  - On a le droit d'utiliser que des méthodes de l'interface
    - Bien qu'elles soient des méthodes abstraites...
- Intérêt : On peut promettre que les objets d'une classe implémentent une certaine fonctionnalité de base
  - Ex : **interface StackLike {**

    Object pop() ;

    void push(Object o) ;

    Object top() ;

    }

Ex : Comparable

- Cas d'usage : déclarer que les objets de notre classe peuvent être comparés

```
interface canCompare { boolean isBigger(Object o); }

class Complex implements canCompare {
        double x,y;
        Complex(double x, double y){ this.x = x; this.y = y; }
        public boolean isBigger(Complex o){
                return (x*x+y*y)>(o.x*o.x+o.y*o.y);
        }
        public String toString() { return x+" i"+y; }

}
```

Ex : Comparable

- Cas d'usage : déclarer que les objets de notre classe peuvent être comparés

```
interface canCompare { boolean isBigger(Object o); }

class Complex implements canCompare {

        double x,y;

        Complex(double x, double y){ this.x = x; this.y = y; }

        public boolean isBigger(Object o1){

                Complex o = (Complex o1) ;
                return (x*x+y*y)>(o.x*o.x+o.y*o.y);

        }

        public String toString() { return x+" i"+y; }

}
```

Ex : Comparable

- Cas d'usage : déclarer que les objets de notre classe peuvent être comparés

```
interface canCompare { boolean isBigger(Object o); }

class Rational implements canCompare {
        int n,d;
        Rational(int n, int d) { this.n = n; this.d = d; }
        public boolean isBigger(Object o1){
                Rational o = (Rational)o1;
                return (n*o.d) > (o.n*d);
        }
        public String toString() { return n+"/"+d; }
}
```

Ex : Comparable

- Cas d'usage : déclarer que les objets de notre classe peuvent être comparés

```
interface canCompare { boolean isBigger(Object o); }

class TestCompare {

        public static canCompare [] sort(canCompare [] tab){

                for(int i=0; i<tab.length; i++)

                        for(int j=0; j<tab.length-1; j++)

                                if(tab[j].isBigger(tab[j+1])){

                                        canCompare tmp = tab[j];

                                        tab[j] = tab[j+1]; tab[j+1] = tmp;  }

                return tab;         }

}
```

Ex : Comparable

- Cas d'usage : déclarer que les objets de notre classe peuvent être comparés

```
interface canCompare { boolean isBigger(Object o); }
class TestCompare {
        public static void main(String [] args){
        canCompare [] tab1 = new Complex [5];
        tab1[0] = new Complex(5.0,0.0);
        tab1[1] = new Complex(3.0,0.0);
        tab1[2] = new Complex(4.0,0.0);
        tab1[3] = new Complex(2.0,0.0);
        tab1[4] = new Complex(1.0,0.0);
        for(Object item:tab1) System.out.println(item);
        tab1 = sort(tab1);
        for(Object item:tab1) System.out.println(item);}
```

Ex : Comparable

- Cas d’usage : déclarer que les objets de notre classe peuvent être comparés

```
interface canCompare { boolean isBigger(Object o); }
class TestCompare {
        public static void main(String [] args){
        canCompare [] tab1 = new Rational [5];
        tab1[0] = new Rational(1,3);
        tab1[1] = new Rational(1,2);
        tab1[2] = new Rational(2,5);
        tab1[3] = new Rational(3,7);
        tab1[4] = new Rational(1,9);
        for(Object item:tab1) System.out.println(item);
        tab1 = sort(tab1);
        for(Object item:tab1) System.out.println(item); }
```

Ex : Comparable

- Cas d'usage : déclarer que les objets de notre classe peuvent être comparés

```
interface canCompare { boolean isBigger(Object o); }
class TestCompare {
        public static void main(String [] args){
        canCompare [] tab1 = new canCompare [5];
        tab1[0] = new Rational(1,3);
        tab1[1] = new Rational(1,2);
        tab1[2] = new Rational(2,5);
        tab1[3] = new Rational(3,7);
        tab1[4] = new Rational(1,9);
        for(Object item:tab1) System.out.println(item);
        tab1 = sort(tab1);
        for(Object item:tab1) System.out.println(item); }
```

Ex : Comparable

- Cas d'usage : déclarer que les objets de notre classe peuvent être comparés

```
interface canCompare { boolean isBigger(Object o); }
class TestCompare {
        public static void main(String [] args){
        canCompare [] tab1 = new canCompare [5];
        tab1[0] = new Rational(1,3);
        tab1[1] = new Rational(1,2);
        tab1[2] = new Complex(2,5);
        tab1[3] = new Rational(3,7);
        tab1[4] = new Rational(1,9);
        for(Object item:tab1) System.out.println(item);
        tab1 = sort(tab1);
        for(Object item:tab1) System.out.println(item); }
```

Comparable

- L’interface qu’on a défini marche correctement, mais fait des comparaisons avec la class **Object**
  - Inévitable : sinon, on ne peut pas définir une version assez générale de l’interface
  - Problème : le compilateur ne peut pas vérifier qu’on utilise les bons types. On est forcé d’utiliser type-casting
- Solution (à voir plus tard) : Type génériques
  - On peut définir l’interface

```
interface canCompare<T>{
        boolean isBigger(T o) ;
}
```

- Cette interface existe déjà en Java :
  - → **Comparable**

Quelques interfaces

- Java a déjà défini quelques interfaces :
  - **Comparable**
  - **Cloneable**
    - La classe implémente la méthode clone() == copie profonde (quel est l'intérêt ?)
  - **Runnable**
    - La classe implémente une méthode (run()) qui peut être exécutée en parallèle. (voir **Threads**)
  - **Serializable**
    - La classe implément une méthode qui permet de sauvegarder ses objets.

Pièges – Héritage Multiple

```
interface interA{ void f(); }
class B{ void h(){ System.out.println("B.h"); }
        int f() { return 2; }
}
class A extends B implements interA{
     ...
```

Pièges – Héritage Multiple

```
interface interA{ void f(); }
class B{ void h(){ System.out.println("B.h"); }
        int f() { return 2; }
}
class A extends B implements interA{
     void f() { … }
```

Pièges – Héritage Multiple

```
interface interA{ void f(); }
class B{ void h(){ System.out.println("B.h"); }
        int f() { return 2; }
}
class A extends B implements interA{
     int f() { … }
```

Abstract !

Pièges – Héritage Multiple

- En cas d’héritage multiple, on peut avoir des conflits
  - Méthodes déclarées avec la même signature !
  - À éviter !
- Si plusieurs déclarations avec même signature et type de retour
  - Correcte. L’implémentation héritée par une classe prend précédence.
- Si plusieurs méthodes héritées par des interfaces
  - Il faut redéfinir (obligatoire!)

Default

- Normalement, les méthodes d'une interface sont **abstract**
- On peut donner une implémentation si on utilise le mot **default**
- Ex :

```
interface StackLike {
        default Object Top(){
          Object o = Pop(); // ?.Pop() ?
          Push(o);
          return o; };
        Object Pop();
        void Push(Object o);
}
```

Héritage Multiple

- Une interface peut étendre (**extends**) une autre
- L'héritage multiple est permis pour les interfaces !
  - Pourquoi ?
  - L'héritage multiple n'est pas permis pour les classes
    - →Ambiguïté : si on hérite deux méthodes avec la même signature, quelle implémentation adopter ?
  - Ce problème n'existe pas pour les interfaces
    - En cas de conflit, même s'il existe une implémentation **default**, la sous-classe/sous-interface **doit redéfinir**.
    - La redéfinition lève l'ambiguïté : la redéfinition est la version principale.

Exemple : FIFO/LIFO/etc

- Regardez l'implémentation d'un conteneur générique qui peut simuler une pile, une file,..., sur la page web.
  - Les fonctionnalités sont définies dans plusieurs fichiers d'interface
    - Un utilisateur peut baser son code sur cette définition
  - Pour plus d'abstraction, on ajout un classe abstraite qui implémente les interfaces
    - Rappel : classe abstraite peut contenir des attributs
  - Pour concrétiser on programme une classe qui étend la classe abstraite.