

# Les interfaces

Maria Virginia Aponte

CNAM-Paris

March 14, 2018

## Interface

- Ensemble de **profils de méthodes** correspondant au **minimum de fonctionnalités requises** dans des classes.
- Mot-clé `interface` et contient:
  - un ensemble d'**entêtes** de méthodes;
  - **aucune implantation** pour les méthodes;
  - **aucune variable**;

# Exemple d'interface

```
public interface Displaceable {  
    public int getX();  
    public int getY();  
    public void move(int dx, int dy);  
}
```

- **Pas de variables** : on ne dit pas *comment* un objet "déplaçable" est construit;
- **Profils de méthodes** : on dit quelles opérations on pourra invoquer sur un objet;
- **Pas de corps de méthodes** : mais on ne dit pas comment elles sont implantées.

# Interfaces: A quoi cela sert?

## A quoi cela sert?

- **Type** : donner le type d'un objet
  - en disant *ce qu'il fait*;
  - et non pas *comment* il le fait,
  - ni comment *il est construit*;
- **Contrat** : «cahier de charges» pour TOUS les objet qui **implament** l'interface.

Interface  $\Rightarrow$  TYPE pour un objet

Basé **UNIQUEMENT** sur ce que l'objet SAIT FAIRE.

# Planter une interface

Une classe **C** implante une interface **I**

- via le mot-clé `implements`
- si **C** fournit au moins autant de méthodes (avec types compatibles) que celles dans l'interface;

⇒ la classe **C** satisfait le «contrat établi» par l'interface **I**.

Flexibilité

**C** peut avoir d'autres méthodes, constructeurs, variables, etc.

# Exemple d'implantation: points déplaçables

```
public class Point implements Displaceable {
    private int x, y;

    public Point(int x0, int y0) {
        x = x0; y = y0;
    }
    public int getX() { return x; }
    public int getY() { return y; }
    public void move(int dx, int dy) {
        x = x + dx; y = y + dy;
    }
}
```

- **en magenta** : les méthodes requises par l'interface;
- *le reste de l'implantation* est non contraint par «le contrat».

# Une autre implantation: cercles déplaçables

- déplacement d'un cercle : implémenté par déplacement de son centre.

```
public class Circle implements Displaceable {
    private Point center;
    private int radius;
    public Circle(Point initCenter, int initRadius) {
        center = initCenter; radius = initRadius;
    }
    public int getRadius() { return radius; }
    public int getX() { return center.getX(); }
    public int getY() { return center.getY(); }
    public void move(int dx, int dy) {
        center.move(dx, dy);
    }
}
```

# La déclaration `implements`

```
class Point implements Displaceable {... }  
class Circle implements Displaceable {...}
```

- Donnée dans l'entête d'une classe (à la déclaration);
- permet d'indiquer que la classe implante l'interface;
- elle est **vérifiée par le compilateur** ⇒  
provoque une erreur si la classe n'implante pas l'interface!

Point et Circle **ont en commun** les méthodes de l'interface.



# Interface = contrat/spécification

Interface I = description **abstraite** d'une classe **quelconque**

- uniquement les profils des méthodes *à implanter*;
- pas de variables, pas de code;
- idée : établir **un lien de typage**
  - entre une interface I et
  - les classes possédant les méthodes «requisés» par I.
- Mais, une interface **n'est pas une classe!**

Avec une interface on **ne peut pas** fabriquer des objets.

# Classe = implantation

Une classe contient toutes les informations nécessaires pour **implanter** des objets:

- les variables d'instance + type ;
- les constructeurs qui initialisent ces variables;
- les méthodes qui agissent sur ces variables;
- l'implantation complète (code) pour ces méthodes.

Avec une classe **on peut fabriquer des objets.**

# Les interfaces sont des types

Type: d'une **variable** :

```
Displaceable d;
```

- cette variable «**Pointe vers**» **implantation** ( vers objet **instance d'une classe qui implante** l'interface)

```
Displaceable d1 = new Point(1,2);  
Displaceable d2 = new Circle(new Point(1,2),3);
```

- **Opérations restreintes** UNIQUEMENT à celles de l'interface :

```
d2.move(-1,1); // OK  
d2.getY();    // donne 3.0  
d2.radius();  // erreur de compilation
```

# Interfaces et abstraction

- *Type unique* pour toute implantation d'objets «déplaçables».
- *code commun* à ces objets  $\Rightarrow$  indépendant de leur «vritable nature».

```
public void moveItAll (ArrayList<Displaceable> s,
                      int dx, int dy) {
    for (int i=0; i < s.size(); i++) {
        s.get(i).move(dx, dy);    }
}

public void exemple () {
    Displaceable p = new Point(5,5);
    Displaceable c = new Circle(new Point(0,0),100);
    ArrayList<Displaceable> lde = new ArrayList<Displaceable>();
    lde.add(p); lde.add(c); // point et cercle
    moveItAll(lde, 5, 10);
}
```

# Mécanisme puissant d'abstraction

Cercle avec centre «déplaçable»  $\Rightarrow$  permet les cercles concentriques.

```
public class Circle implements Displaceable {
    private Displaceable center;
    private int radius;
    public Circle(Displaceable c, int r) {
        center = c; radius = r;
    }
    public int getRadius() { return radius; }
    public int getX() { return center.getX(); }
    public int getY() { return center.getY(); }
    public void move(int dx, int dy) {
        center.move(dx, dy);
    }
}
```

# Quelle méthode `move` est exécutée?

La méthode `move` exécutée est celle de l'objet courant!

```
public static void main(String[] args) {  
    Point p = new Point(1,1);  
    Circle c1 = new Circle(p, 3);  
    Circle c2 = new Circle(c1, 10);  
    c2.move(2, 2);  
}
```

Interfaces : *description abstraite* du comportement des objets.

- les bibliothèques Java  $\Rightarrow$  opérations très génériques sur objets décrits par une interface;
- on peut les utiliser avec n'importe quelle classe qui les implante;
- ces bibliothèques deviennent alors *hautement réutilisables*.

## Interfaces et bibliothèques Java

Indispensable: connaître les interfaces pour employer les bibliothèques!

Le développement de logiciels de grande taille

- se fait souvent par plusieurs équipes, en parallèle;
- les objets conçus par l'équipe A pourront utiliser des objets conçus par l'équipe B;
- comment s'accorder sur ce que doit faire chaque classe écrite par des équipes différentes?

Aide: établir des contrats précis pour chaque classe;

⇒ utiliser des interfaces.



Une interface pour les modules d'enseignement:

```
public interface avecNote{  
    String getCodeModule();  
    double calculeNote();  
  
}
```

Elle possède:

- une méthode pour obtenir le nom du module;
- une méthode de calcul de la note du module;
- aucune variable, pas d'implantation des méthodes.
- les profils des méthodes **ne doivent pas** être spécifiés

public

# Classe AlgoProg2

---

```
public class AlgoProg2 {
    double session1;
    double session2;
    String code = "NFA002";
    public AlgoProg2(double s1, double s2) {
        session1 = s1; session2 = s2;
    }
    public String getCodeModule() {return code;}

    public double calculeNote() {
        if (session1 >= 10 || session2 == 0) return session1
        else return session2;
    }

    public void afficher(){...}
}
```

---

# Classe AlgoProg5

---

```
public class AlgoProg5 {
    double projet1;
    double projet2;
    String code = "NFA005";
    public AlgoProg5(double p1, double p2) {
        projet1 = p1; projet2 = p2;
    }
    public String getCodeModule() {return code;}

    public double calculeNote() {
        return (projet1 + projet2)/2;
    }
}
```

---

# Interfaces: «commun dénominateur»

Les classes `AlgoProg2` et `AlgoProg5` possèdent:

- toutes les **fonctionnalités** de l'interface `AvecNote`:
  - méthode `getCodeModule` (même profil et publique);
  - méthode `calculeNote` (même profil et publique);
- `AlgoProg2` possède **en plus**: méthode `afficher`;
- variables + code des 2 classes **sont différents**.

`AlgoProg2` et `AlgoProg5` ont en commun **toutes** les méthodes de l'interface.

# Interface Affichable

---

```
interface Affichable{
    void afficher();
}

class Compte implements Affichable {
    private int solde;
    public Compte(int init) { solde = init; }
    public void depot(int n){ solde = solde+n; }
    public void retrait(int n){ solde = solde-n; }
    public void afficher(){ Terminal.ecrireStringln("Sold
}
```

---

La classe Compte **implante** l'interface Affichable

# AlgoProg2 qui implante plusieurs interfaces

```
class AlgoProg2 implements AvecNote, Affichable {  
    double session1; double session2;  
    String code = "NFA002";  
    public AlgoProg2(double s1, double s2) {  
        session1 = s1; session2 = s2;  
    }  
    public String getCodeModule() {return code;}  
  
    public double calculeNote() {  
        if (session1 >= 10 || session2 == 0) return session1  
        else return session2;  
    }  
  
    public void afficher(){...}  
}
```

AlgoProg2 **implante** AvecNote **et** Affichable.

# Bilan: ce qu'on ne peut pas faire avec une interface

- 1 **donner des détails d'implantation:**
  - pas de variables,
  - pas de code pour donner l'algorithme des méthodes;
- 2 **fabriquer des objets** : on ne saurait pas quoi mettre dedans (variables), ni quoi exécuter (code).

# Bilan: à quoi sert une interface?

- 1 **spécification de méthodes**: de toute classe voulant l'**implanter**;
- 2 **dire ce qui est commun** à toutes les classes qui la déclarent;
- 3 **Pour 1 classe C**: dire toutes les **interfaces** qu'elle implante;
- 4 **typage**: pour déclarer des variables de (type) de l'interface;
- 5 **programmation abstraite**
- 6 Utilisées pour décrire les classes d'une bibliothèque (API).



# 1,2: spécification de contrat de méthodes

- 1 spécification de "contrat": pour toute classe voulant **implanter** l'interface;
- 2 spécification commune à plusieurs classes (avec `implements`) pour chacune des classes;

---

```
interface AvecNote { //1: Spec de contrat
    double calculeNote();
    String getCodeModule();
}
```

```
class AlgoProg2 implements AvecNote //2: méthodes commu
```

```
class AlgoProg5 implements AvecNote //2: méthodes commu
```

---

### 3. Déclarer toutes les interfaces qu'une classe implante

Si une classe  $C$  implante plusieurs interfaces  $I_1, \dots, I_n$  cela revient à dire:

- qu'elle est conforme à tous ces "contrats";
- autrement dit, qu'elle contient au moins les méthodes de toutes ces interfaces.

---

```
class AlgoProg2 implements AvecNote, Affichable {...
```

---

## 4. Les interfaces comme types

On peut employer une interface **en tant que type**:

- pour **déclarer le type** de variables ou de paramètres,
- avec pour type le nom de l'interface;

---

```
AvecNote m1, m2;  
Affichable o1, o2;  
....  
static double moyenne(AvecNote m1, m2) {  
    return (m1.calculerNote() + m2.calculerNote())/2;  
}
```

---

## 4. Que peut-on faire avec une variable de type interface?

Si une variable  $x$  est déclarée avec le type d'une interface  $I$ :

Si on déclare: `I x;`

- on ne peut affecter à  $x$  qu'un objet  $o$  instance d'une classe  $C$  qui implante  $I$ ;
- on ne peut invoquer sur  $x$  que les méthodes de l'interface  $I$ ;
- les autres méthodes de  $C$ , ne sont pas accessibles dans  $x$ ;
- mais elles sont toujours accessibles dans  $o$ .

# Affectation sur un variable de "type interface"

```
I x = new C();
```

- si `x` est déclaré du type d'une interface `I`,
- et si `class C implements I{...}`,
- alors on peut affecter `x` avec tout objet instance de `C`.

---

```
AvecNote x1, x2;           // type  
x1 = new AlgoProg2(10, 0); // affectation instance Algo  
x2 = new AlgoProg5(10, 7); // affectation instance Algo
```

---

# Invocations sur objet de "type interface"

Invocation d'une méthode `m` sur l'objet `x`: `x.m()`

- Supposons `x` déclaré de type `I`,
- et `class C implements I{...}`,
- et `C o = new C();` (un objet `o` de type `C`),
- et `x = o;` (`x` contient l'objet `o`),
- on peut invoquer `x.m()` **seulement** si `m` est une méthode de `I`.

---

```
AvecNote x1,x2;           // declaration
x1 = new AlgoProg2(10,0); // instance AlgoProg2
x2 = new AlgoProg5(10,7); // instance AlgoProg5
note1 = x1.calculNote(); // calculNote() dans AvecNote
x1.afficher();           // erreur: afficher pas dans AvecNote
```

---

## Invocations sur objet de "type interface" (2)

- Si `x` déclaré de type `I`, et `class C implements I`,
- et `C o = new C();` (un objet `o` de type `C`),
- et `x = o;` (`x` contient l'objet `o`),
- on peut invoquer `x.m()` **seulement** si `m` est une une méthode de `I`.

---

```
AvecNote x;  
AlgoProg2 a = new AlgoProg2(10,0);  
x = a; // instance AlgoProg2  
note1 = x1.calculerNote(); // calculerNote() dans AvecNote  
x.afficher(); // erreur: afficher pas dans AvecNote  
a.afficher() // ok, afficher visible dans a
```

---

# Invocations sur objet de "type interface" (3)

- `x` et `a` pointent vers le même objet en mémoire (instance de `AlgoProg2`),
- mais, `x` et `a` ont des types différents:
- `a` de type `AlgoProg2`: ce qui rend "disponibles" dans `a` **seulement** les méthodes de cette classe;
- `x` de type `AvecNote`: ce qui rend "disponibles" dans `x`, **seulement** les méthodes de cette interface;

Si `x` est déclaré de type `T`:

- le type `T` agit comme un "filtre" sur `x`,
- on ne pourra invoquer sur `x` que ce qui est décrit par `T`,
- indépendamment de ce qui peut être effectivement pointé par `x`.



## 5. Programmation abstraite

On peut employer les interfaces pour:

- écrire des programmes qui s'appuient exclusivement sur les fonctionnalités présentes dans l'interface des objets qu'ils manipulent,
- et pas sur d'autres détails d'implantation (variables, algorithmes, autres méthodes).
- On obtient des programmes qui fonctionnent sur toute classe qui implante l'interface,
- donc, qui fonctionnent sur des classes de types différents!

## 5. Programmation abstraite (exemple)

- Des programmes qui fonctionnent sur toute classe qui implante une interface,
- donc, qui fonctionnent sur des classes de types différents!

---

```
static double moyenne(AvecNote m1, m2) {  
    return (m1.calculerNote() + m2.calculerNote())/2;  
}  
....  
AlgoProg2 a = new AlgoProg2(10,0);  
AlgoProg5 b = new AlgoProg5(10,7);  
  
double m = moyenne(a,b);
```

---

Pourquoi cet appel est correct?

## 5. Programmation abstraite (exemple)

- On peut également fabriquer des structures de données dont les composantes sont typées par une interface,
- et qui peuvent donc contenir des objets de classes différentes!

---

```
AlgoProg2 a = new AlgoProg2(10,0);
AlgoProg5 b = new AlgoProg5(10,7);
Compte c = new Compte();
AvecNote [] tn = {a,b};
Affichable [] ta = {a,c};
double somme = 0;
for (int i=0; i<tn.length; i++){
    somme = somme + tn[i].calculeNote();
}
for (int i=0; i<ta.length; i++){
    ta[i].afficher();
}
```

- Mécanisme de description des classes dans les bibliothèques contenant des programmes utilitaires ou *boîtes à outils*.
- Dans la documentation du langage, chaque boîte à outil est décrite par les noms de ses sous-programmes leurs types de retour, et les types de leurs paramètres: c'est une interface!

*API* ou *Applications Programming Interface* est la description ou cahier de charges d'une boîte à outils.

- Il s'agit des informations d'une interface,
- plus une description textuelle des méthodes,
- plus une description des autres éléments de la classe: variables, constructeurs, etc.