Interfaces Java

Virginia Aponte

CNAM-Paris

8 septembre 2023

Idée derrière la notion d'interface

Répondre manière générique à des questions du genre :

Quelles fonctionnalités doit proposer un objet qui implante un concept du programme, p.e. : « banque d'un client »

Réponse en Java via la notion de Interface

- Spécifier de la manière la plus générale possible (c.a.d., sans donner d'implantation) les méthodes que les futurs objets qui implantenront ce concept devront posséder,
- faire en sorte que le compilateur vérifie qu'une classe donnée correspond bien à cet interface.

Idée derrière la notion d'interface

En pratique :

1. quelles méthodes (avec quels paramètres, types) devront posséder ces objets ?
2. pas d'implantation imposée $\Rightarrow$ on ne dit rien sur leurs variables d'instance
3. on veut que la correspondance entre spécification et objet qui l'implante soit vérifiée par le compilateur.

Interfaces en Java : définition

Interface

- Ensemble de profils de méthodes correspondant au minimum de fonctionnalités requises dans n'importe quel classe « implantation ».
- Mot-clé `interface` avec :
  - un ensemble d'entêtes de méthodes ;
  - aucune implantation pour les méthodes ;
  - aucune variable ;

Exemple : interface « banque d'un client »

- On suppose qu'un client possède un seul compte.

```
public interface IBanqueClient {
    double getSolde(int num);
    boolean depot(int num,  double m);
    boolean retrait(int num,  double m);
    boolean estBloque(int num);
}
```

Exemple : interface « banque d'un client »

- On suppose qu'un client possède un seul compte.
- Il ne le manipule pas directement $\Rightarrow$ numéro de compte pour éffectuer les opérations.

```
public interface IBanqueClient {
    double getSolde(int num);
    boolean depot(int num,  double m);
    boolean retrait(int num,  double m);
    boolean estBloque(int num);
}
```

Exemple : interface « banque d'un client »

- On suppose qu'un client possède un seul compte.
- Il ne le manipule pas directement $\Rightarrow$ numéro de compte pour éffectuer les opérations.
- Objet banque possède beaucoup d'opérations : créer nouveau compte, bloquer un compte, etc.

```
public interface IBanqueClient {
    double getSolde(int num);
    boolean depot(int num,  double m);
    boolean retrait(int num,  double m);
    boolean estBloque(int num);
}
```

Exemple : interface « banque d'un client »

- On suppose qu'un client possède un seul compte.
- Il ne le manipule pas directement $\Rightarrow$ numéro de compte pour éffectuer les opérations.
- Objet banque possède beaucoup d'opérations : créer nouveau compte, bloquer un compte, etc.
- Banque d'un client restreinte aux opérations sur son compte.

```
public interface IBanqueClient {
    double getSolde(int num);
    boolean depot(int num,  double m);
    boolean retrait(int num,  double m);
    boolean estBloque(int num);
}
```

Exemple : classe implantation

- `Banque` réunit tous les comptes dans une liste, et implante toutes les opérations de la banque (10 en tout, voir démo))

```
public class Banque implements IBanqueClient, IBanque {
  private ArrayList<ICompte> cpts;
  public double getSolde(int n){ // de IBanqueClient
        ICompte c = getCompteDeNum(n);
        return c.getSolde();
  }
  public void bloquer(int n) { // de IBanque
        ICompte c = getCompteDeNum(n);
        c.bloquer();
  }
 ...
```

Exemple : classe implantation

- `Banque` réunit tous les comptes dans une liste, et implante toutes les opérations de la banque (10 en tout, voir démo))
- Elle implante les interfaces `IBanqueClient`, et `IBanque`,

```
public class Banque implements IBanqueClient, IBanque {
  private ArrayList<ICompte> cpts;
  public double getSolde(int n){ // de IBanqueClient
        ICompte c = getCompteDeNum(n);
        return c.getSolde();
  }
  public void bloquer(int n) { // de IBanque
        ICompte c = getCompteDeNum(n);
        c.bloquer();
  }
 ...
```

Exemple : classe implantation

- `Banque` réunit tous les comptes dans une liste, et implante toutes les opérations de la banque (10 en tout, voir démo))
- Elle implante les interfaces `IBanqueClient`, et `IBanque`,
- `getCompteDeNum(n)` renvoie le compte avec numéro n ou `null` si non trouvé.

```
public class Banque implements IBanqueClient, IBanque {
  private ArrayList<ICompte> cpts;
  public double getSolde(int n){ // de IBanqueClient
        ICompte c = getCompteDeNum(n);
        return c.getSolde();
  }
  public void bloquer(int n) { // de IBanque
        ICompte c = getCompteDeNum(n);
        c.bloquer();
  }
 ...
```

Une classe qui implante une interface

Une classe implante une interface

- si elle fournit au moins autant de méthodes (avec types compatibles) que celles requises par l'interface ;
- signalé par le mot-clé `implements`

dans ce cas on dit que :

$\Rightarrow$ la classe satisfait le contrat établit par l'interface

c'est vérifié par le compilateur !

Erreur de compilation si la classe marquée `implements` viole cette correspondance.

Interfaces : un outil de spécification(suite)

- interface = types
  - on peut déclarer un objet c du type d'une interface IC
  - puis appliquer des méthodes de l'interface IC à cet objet !

```
public interface IC { .... } //une interface
public class Exemple {
   IC c;    // c est de type interface IC
   c.m1(); // m1 est dans IC
   ...

public class C implements IC {....}
```

  - ce code compilera !

- Développement séparé on peut développer le code des classes C et Exemple de manière indépendante !

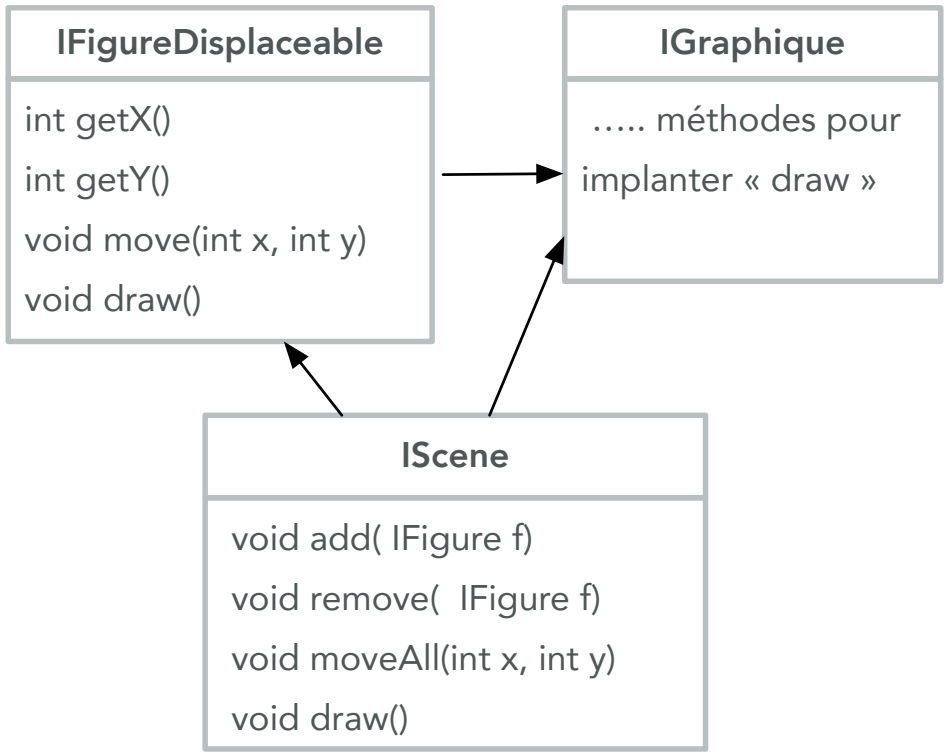
Interfaces et développement séparé (1)

Exemple :

- Écrire une application d'animation de figures géométriques dans une scène.
- Pour une scène on pourra ajouter, retirer ou déplacer des figures.
- On souhaite repartir le travail entre plusieurs équipes de développeurs qui travaillent en parallèle.

Interfaces et développement séparé (2)

Extrait d'architecture décrivant les interfaces et de leurs dépendances. $IA \rightarrow IB$ signifie : la classe qui implante l'interface IA utilise des méthodes de la classe qui implante IB.



Interfaces et développement séparé (3)

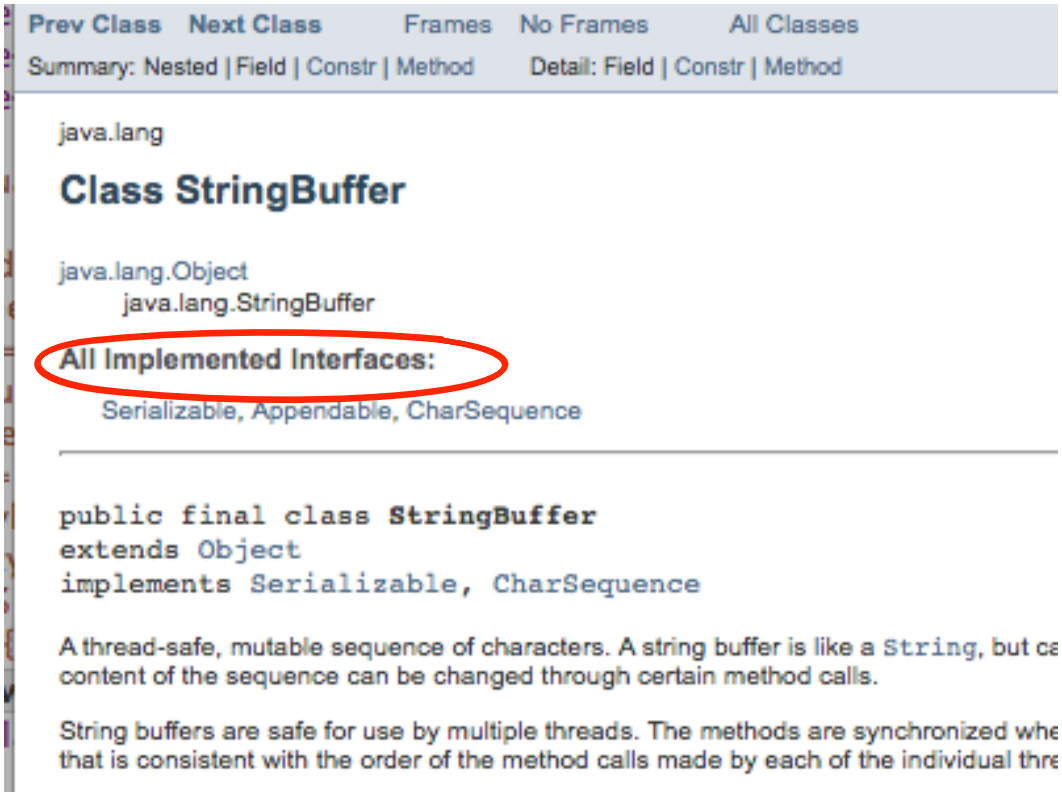
Solution :

1. Concevoir l'architecture d'interfaces pour les classes :
   - chacune avec liste de méthodes+ contrats
2. Chaque sous-équipe développe une ou plusieurs classes
   - l'appel à une méthode développée par une autre équipe se fait sur la base de ce que disent les contrats-interfaces ;
   - Pendant le développement, le code écrit séparemment compile, car objets(+méthodes) sont déclarés du type des interfaces.
   - L'exécution pourra se faire seulement lorsque les implantations des interfaces sont terminées.

Interfaces et bibliothèques Java

En pratique

Les bibliothèques Java sont bâties sur diverses interfaces : on doit savoir les lire + utiliser.



Prev Class Next Class Frames No Frames All Classes

Summary: Nested | Field | Constr | Method Detail: Field | Constr | Method

java.lang

Class StringBuffer

java.lang.Object
    java.lang.StringBuffer

All Implemented Interfaces:

Serializable, Appendable, CharSequence

```
public final class StringBuffer
extends Object
implements Serializable, CharSequence
```

A thread-safe, mutable sequence of characters. A string buffer is like a `String`, but ca
content of the sequence can be changed through certain method calls.

String buffers are safe for use by multiple threads. The methods are synchronized whe
that is consistent with the order of the method calls made by each of the individual thre

Exemple : interface `Comparable` (bibliothèque Java)

En Java, les opérations de tri sont :

- définies sur les objets de n'importe quel type `T` (objet),
- à condition qu'ils soient comparables entre eux (selon un ordre).

Ces objets doivent implanter l'interface `Comparable<T>` :

```
interface Comparable<T> {
  /**
   * Retourne un entier:
        -  positif si this est plus grand que o,
        -  0 si egaux,
        -  negatif sinon
   */
  int compareTo(T o);
}
```

Les interfaces sont des contrats

Une interface spécifie :

- les méthodes qu'un objet doit posséder ;

Cohérence interne $\Rightarrow$ on parle d'invariant d'état.

Les interfaces sont des contrats

Une interface spécifie :

- les méthodes qu'un objet doit posséder ;
- le comportement de ces méthodes (javadoc, pré et post conditions) ;

Cohérence interne $\Rightarrow$ on parle d'invariant d'état.

Les interfaces sont des contrats

Une interface spécifie :

- les méthodes qu'un objet doit posséder ;
- le comportement de ces méthodes (javadoc, pré et post conditions) ;
- éventuellement une *propriété invariante de l'état interne* : sans cela l'objet peut devenir incohérent.

Cohérence interne $\Rightarrow$ on parle d'invariant d'état.

Les interfaces sont des contrats

Une interface spécifie :

- les méthodes qu'un objet doit posséder ;
- le comportement de ces méthodes (javadoc, pré et post conditions) ;
- éventuellement une *propriété invariante de l'état interne* : sans cela l'objet peut devenir incohérent.
  - interface pour les dates : la date interne est toujours correcte ;

Cohérence interne $\Rightarrow$ on parle d'invariant d'état.

Les interfaces sont des contrats

Une interface spécifie :

- les méthodes qu'un objet doit posséder ;
- le comportement de ces méthodes (javadoc, pré et post conditions) ;
- éventuellement une *propriété invariante de l'état interne* : sans cela l'objet peut devenir incohérent.
  - interface pour les dates : la date interne est toujours correcte ;
  - interface pour les télé-cabines : une cabine en mouvement a toujours les portes verouillées.

Cohérence interne $\Rightarrow$ on parle d'invariant d'état.

Les interfaces sont des contrats

Une interface spécifie :

- les méthodes qu'un objet doit posséder ;
- le comportement de ces méthodes (javadoc, pré et post conditions) ;
- éventuellement une *propriété invariante de l'état interne* : sans cela l'objet peut devenir incohérent.
  - interface pour les dates : la date interne est toujours correcte ;
  - interface pour les télé-cabines : une cabine en mouvement a toujours les portes verouillées.
- toute implantation *doit maintenir* la cohérence interne des objets ;

Cohérence interne $\Rightarrow$ on parle d'invariant d'état.

Exemple 2 : Interface pour objets déplaçables

Les méthodes qu'un objet déplaçable doit posséder :

- obtenir sa position courante (`getX()`, `getY()`);

Cette spécification permet toutes sorte d'implantations :

```
public interface Displaceable {
 /**  Obtenir abscisse (position). */
  public int getX();
 /**  Obtenir ordonnée (position). */
  public int getY();
 /**  Déplacement avec differentiel (x+dx,y+dy)  */
  public void move(int dx, int dy);
}
```

Exemple 2 : Interface pour objets déplaçables

Les méthodes qu'un objet déplaçable doit posséder :

- obtenir sa position courante (`getX()`, `getY()`);
- changer cette position (`move`);

Cette spécification permet toutes sorte d'implantations :

```
public interface Displaceable {
 /**  Obtenir abscisse (position). */
  public int getX();
 /**  Obtenir ordonnée (position). */
  public int getY();
 /**  Déplacement avec differentiel (x+dx,y+dy)  */
  public void move(int dx, int dy);
}
```

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;
  }
}
```

- Spécifie les objets Point qui sont déplaçables ;
- en magenta : les méthodes requises par l'interface
- *le reste de l'implantation* est "laissée libre" par le contrat.

Autre exemple d'implantation : cercles deplaçables

- Spécifie les cercles déplaçables ;
- `move` implanté par le déplacement du centre du cercle ;

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

Types-interfaces : restriction d'accès

- Type des variables : nous pouvons déclarer une variable avec une interface en lieu et place de son type ;

```
Displaceable d;
```

Types-interfaces : restriction d'accès

- Type des variables : nous pouvons déclarer une variable avec une interface en lieu et place de son type ;

```
Displaceable d;
```

- Implantation : elle pourra prendre la valeur de n'importe quel objet d'une classe qui `implements` l'interface

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

Types-interfaces : restriction d'accès

- Type des variables : nous pouvons déclarer une variable avec une interface en lieu et place de son type ;

```
Displaceable d;
```

- Implantation : elle pourra prendre la valeur de n'importe quel objet d'une classe qui `implements` l'interface

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

- Seules les opérations de l'interface son appelables via ces variables

```
d2.move(-1,1);
d2.getX();
d2.getRadius(); // erreur de compilation:
                // getRadius absente dans Diplaceable
```

Interfaces et abstraction

Une scène animée est composée d'une liste de figures déplaçables.

- la classe Scene contient :

```
public class Scene {
 ArrayList<Displaceable> scene; //figures de la scene
 public void moveAll (int dx, int y) {
   for (int i=0; i < s.size(); i++) {
      scene.get(i).move(dx,dy);
   }}
 public add(Displaceable fig) { scene.add(fig); }
}
```

Interfaces et abstraction

Une scène animée est composée d'une liste de figures déplaçables.

- la classe Scene contient :
  - une liste de figures (diverses) déplaçables ;

```
public class Scene {
 ArrayList<Displaceable> scene; //figures de la scene
 public void moveAll (int dx, int y) {
   for (int i=0; i < s.size(); i++) {
      scene.get(i).move(dx,dy);
   }}
 public add(Displaceable fig) { scene.add(fig); }
}
```

Interfaces et abstraction

Une scène animée est composée d'une liste de figures déplaçables.

- la classe Scene contient :
  - une liste de figures (diverses) déplaçables ;
  - une méthode add pour ajouter une figure à la scène ;

```
public class Scene {
 ArrayList<Displaceable> scene; //figures de la scene
 public void moveAll (int dx, int y) {
   for (int i=0; i < s.size(); i++) {
      scene.get(i).move(dx,dy);
   }}
 public add(Displaceable fig) { scene.add(fig); }
}
```

Interfaces et abstraction

Une scène animée est composée d'une liste de figures déplaçables.

- la classe Scene contient :
  - une liste de figures (diverses) déplaçables ;
  - une méthode add pour ajouter une figure à la scène ;
  - une méthode moveAll pour bouger toutes les figures ;

```
public class Scene {
 ArrayList<Displaceable> scene; //figures de la scene
 public void moveAll (int dx, int y) {
   for (int i=0; i < s.size(); i++) {
      scene.get(i).move(dx,dy);
   }}
 public add(Displaceable fig) { scene.add(fig); }
}
```

Interfaces et abstraction

Une scène animée est composée d'une liste de figures déplaçables.

- la classe Scene contient :
  - une liste de figures (diverses) déplaçables ;
  - une méthode add pour ajouter une figure à la scène ;
  - une méthode moveAll pour bouger toutes les figures ;
  - ce code est « générique » : il invoque `move` sur chaque figure sans s'occuper de sa "veritable nature" ;

```
public class Scene {
 ArrayList<Displaceable> scene; //figures de la scene
 public void moveAll (int dx, int y) {
   for (int i=0; i < s.size(); i++) {
      scene.get(i).move(dx,dy);
   }}
 public add(Displaceable fig) { scene.add(fig); }
}
```

Interfaces et abstraction

Une scène animée est composée d'une liste de figures déplaçables.

- la classe Scene contient :
  - une liste de figures (diverses) déplaçables ;
  - une méthode add pour ajouter une figure à la scène ;
  - une méthode moveAll pour bouger toutes les figures ;
  - ce code est « générique » : il invoque `move` sur chaque figure sans s'occuper de sa "veritable nature" ;
- `Displaceable` : *type unique* pour toutes sortes de figures de la scène.

```
public class Scene {
 ArrayList<Displaceable> scene; //figures de la scene
 public void moveAll (int dx, int y) {
   for (int i=0; i < s.size(); i++) {
      scene.get(i).move(dx,dy);
   }}
 public add(Displaceable fig) { scene.add(fig); }
}
```

Interfaces et abstraction (2)

Création d'une scène puis un pas d'animation :

- la variable `anim` est de type Scene.

```
public void exemple () {
    Displaceable s1 = new Point(5,5);
    Displaceable s2 = new Circle(new Point(0,0),100);
    Scene anim = new Scene();
    anim.add(s1);  anim.add(s2);
    anim.moveAll(1,5,10);
 }
```

Interfaces et abstraction (2)

Création d'une scène puis un pas d'animation :

- la variable `anim` est de type Scene.
- on ajoute des figures diverses dans la scène ;

```
public void exemple () {
    Displaceable s1 = new Point(5,5);
    Displaceable s2 = new Circle(new Point(0,0),100);
    Scene anim = new Scene();
    anim.add(s1);  anim.add(s2);
    anim.moveAll(1,5,10);
  }
```

Interfaces et abstraction (2)

Création d'une scène puis un pas d'animation :

- la variable `anim` est de type Scene.
- on ajoute des figures diverses dans la scène ;
- l'appel `anim.moveAll(1,5,10)` les déplace toutes

```
public void exemple () {
   Displaceable s1 = new Point(5,5);
   Displaceable s2 = new Circle(new Point(0,0),100);
   Scene anim = new Scene();
   anim.add(s1);  anim.add(s2);
   anim.moveAll(1,5,10);
 }
```

Interfaces : utilité (résumé)

A quoi cela sert ?

- Type : donner le type d'un objet
  - décrit par *la liste de métodes requises pour ce type d'objet* ;
  - Permet de restreindre la visiblité des méthodes de l'objet aux seules présentes dans l'interface
- Contrat pour les implantations/documentation/tests :
  - spécifier un contrat avec un commentaire javadoc pour chaque méthode de l'interface (javadoc) : servira aux développeurs ET aux testeurs.
  - *documentation*
- Modularité/Abstraction : séparer contrat et implantation ; autoriser le développement séparé, diverses implantation ;