ITI 1521. Introduction à l'informatique II

Interface : type abstrait de données (TAD) et ses implémentations

by

Marcel Turcotte

Version du 19 janvier 2020

Préambule

Préambule

Aperçu

Aperçu

Interface : type abstrait de données (TAD) et ses implémentations

Les déclarations de classes sont l'un des dispositifs de Java pour créer de nouveaux types de données. Dans ce cas-ci, nous disons qu'il s'agit d'un type concret de données. Dans ce module, nous abordons le concept de l'interface qui permettra la définition de types abstraits de données.

Objectif général :

- Cette semaine, vous serez en mesure de déclarer un type abstrait de données par le biais d'une interface.

Une vidéo d'introduction :

- `https://www.youtube.com/watch?v=jUJLiVnCgwo`

Préambule

Objectifs d'apprentissage

Objectifs d'apprentissage

- **Expliquer** dans vos propres mots le concept d'interface.
- **Déclarer** une interface.
- **Implémenter** une interface.

Lectures :

- Pages 2–7 de E. Koffman et P. Wolfgang.

Préambule

Plan du module

Plan

Résumé

Définir un nouveau type

- Une **déclaration de classe** définit un nouveau type

Définir un nouveau type

- La simple déclaration suivante définit un nouveau type

```
public class Point {

}
```

Placée dans un fichier nommé **Point.java**, cette déclaration de classe peut être compilée.

```
> javac Point.java
```

Définir un nouveau type

- On peut alors déclarer une variable de type **Point**

```
public class Test {
    public static void main(String[] args) {
        Point p;
    }
}
```

Définir un nouveau type

- On peut même créer un objet de la classe **Point**.

```
public class Test {
    public static void main(String[] args) {
        Point p;
        p = new Point();
    }
}
```

- On peut le faire parce qu'il existe un **constructeur par défaut**.
- Bien sûr, ces objets n'ont ni variables, ni méthodes d'instances (pour l'instant).

```java
public class Point {

    private double x;
    private double y;

    public Point(double xInit, double yInit) {
        x = xInit;
        y = yInit;
    }
    public double getX() {
        return x;
    }
    // ...
    public void translate(double deltaX, double deltaY) {
        x = x + deltaX;
        y = y + deltaY;
    }
}
```

- Le site du cours vous propose une **implémentation complète**

Utiliser ce nouveau type

```
public class Test {
    public static void main(String[] args) {

        Point p1, p2;

        p1 = new Point(10, 20);
        p2 = new Point(522, 43);

        if (p1.getX() < p2.getX()) {
            p1.translate(p2.getX() - p1.getX(), 0.0);
        }

        if (p1.getY() < p2.getY()) {
            p1.translate(0.0, p2.getY() - p1.getY());
        }

    }
}
```

Définition : type concret de données

- La déclaration de classe définit un **type concret de données**.
- On dit **concret** parce qu'on en définit la représentation des données ainsi que l'implémentation des méthodes.

Interface

Introduction

- Revisitons les deux implémentations de la classe **Pair** présentée au dernier cours.

```java
public class PairVar {

    private int first;
    private int second;

    public PairVar(int firstInit, int secondInit) {
        first = firstInit;
        second = secondInit;
    }
    public int getFirst() {
        return first;
    }
    public int getSecond() {
        return second;
    }
    public void setFirst(int value) {
        first = value;
    }
    public void setSecond(int value) {
        second = value;
    }
}
```

```
public class PairArray {

    private int[] elems;

    public PairArray(int first, int second) {
        elems = new int[2];
        elems[0] = first;
        elems[1] = second;
    }
    public int getFirst() {
        return elems[0];
    }
    public int getSecond() {
        return elems[1];
    }
    public void setFirst(int value) {
        elems[0] = value;
    }
    public void setSecond(int value) {
        elems[1]= value;
    }
}
```

Introduction

- Les classes **PairVar** et **PairArray** sont **deux implémentations possibles d'un même concept**, une paire d'entiers.
- Java possède une construction pour formaliser cette idée.

Interface Pair

```
public interface Pair {

    public abstract int getFirst();

    public abstract int getSecond();

    public abstract void setFirst(int first);

    public abstract void setSecond(int first);

}
```

Interface Pair

- La déclaration d'une interface ressemble à celle d'une classe.
- Elle commence par le mot clé **interface** (et non **class**), suivi d'un identificateur (le nom de l'interface), suivi par le corps de l'interface.

```
public interface Pair {
    public abstract int getFirst();
    public abstract int getSecond();
    public abstract void setFirst(int first);
    public abstract void setSecond(int first);
}
```

La définition est mise dans un fichier du même nom que l'interface, puis elle est compilée afin de produire un fichier **.class**.

Interface

L'**interface** contient :

- Constantes ;
- Méthodes abstraites.

Interface

- Un méthode **abstraite** n'a pas d'implémentation.
- Comme toutes les méthodes de l'interface doivent être abstraites et de visibilité **public**, on peut omettre **public abstract**.

```
public interface Pair {
    int getFirst();
    int getSecond();
    void setFirst(int first);
    void setSecond(int first);
}
```

L'**interface** c'est un contrat. Une liste de méthodes à implémenter.

Type abstrait de données

- Nous avons défini un **type abstrait de données (TAD)**. Nous avons spécifié les **opérations**, mais pas l'**implémentation** !

```
public interface Pair {
    int getFirst();
    int getSecond();
    void setFirst(int first);
    void setSecond(int first);
}
```

Type abstrait de données

- **À quoi** ça sert ?

On peut déclarer une variable de type **Pair** !

```
Pair p;
```

Les opérations **p.getFirst()**, **p.getSecond()**, **p.setFirst(10)**, et **p.setSecond(55)** sont valides.

Type abstrait de données

- Mais, que mettons dans la variable référence **p** ?

```
Pair p;
```

Java : implements

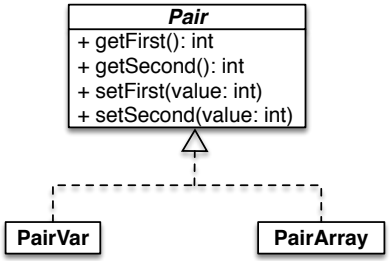
- Nouveau mot-clé, **implements**. Nous dirons que la classe **PairVar réalise** l'interface **Pair**.

```
public class PairVar implements Pair {

    private int first;
    private int second;

    public PairVar(int firstInit, int secondInit) {
        first = firstInit;
        second = secondInit;
    }
    public int getFirst() {
        return first;
    }
    // ...
}
```

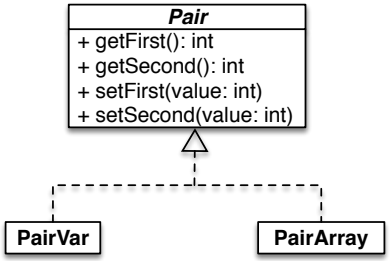
```java
public class PairVar implements Pair {

    private int first;
    private int second;

    public PairVar(int firstInit, int secondInit) {
        first = firstInit;
        second = secondInit;
    }
    public int getFirst() {
        return first;
    }
    public int getSecond() {
        return second;
    }
    public void setFirst(int value) {
        first = value;
    }
    public void setSecond(int value) {
        second = value;
    }
}
```

```java
public class PairArray implements Pair {

    private int[] elems;

    public PairArray(int first, int second) {
        elems = new int[2];
        elems[0] = first;
        elems[1] = second;
    }
    public int getFirst() {
        return elems[0];
    }
    public int getSecond() {
        return elems[1];
    }
    public void setFirst(int value) {
        elems[0] = value;
    }
    public void setSecond(int value) {
        elems[1]= value;
    }
}
```

UML : Pair


Pair
+ getFirst(): int
+ getSecond(): int
+ setFirst(value: int)
+ setSecond(value: int)
PairVar
PairArray

À quoi ça sert ?




```
Pair p;

p = new PairVar(10, 20);

p = new PairArray(10, 20);
```

new Pair() n'est pas valide et c'est logique !

Comparable

Exemple complet

On souhaite implémenter un **algorithme de tri**.

- Comparer un algorithme de tri pour des objets de la classe **Person** et des objets de la classe **Time**.
- Quelles sont les **ressemblances** et les **différences** ?

Algorithme de tri

- L'**algorithme de tri** est le même, peu importe les éléments du tableau.
- Ce qui change, c'est la façon de **comparer les éléments** !

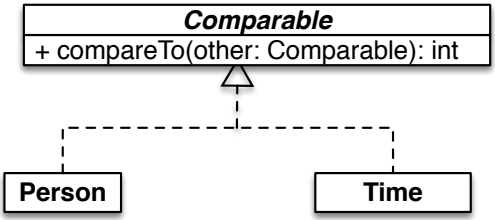
Comparable

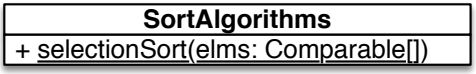
Définissons l'interface **Comparable** :

```
public interface Comparable {
    public int compareTo(Comparable other);
}
```

Un objet est «**Comparable**» s'il possède une méthode de **compareTo** !

UML : Comparable





SortAlgorithms.selectionSort

```
public class SortAlgorithms {
    public static void selectionSort(Comparable[] a) {
        for (int i = 0; i < a.length; i++) {
            int min = i;
            // trouvez l'element le plus petit dans la
            // portion non triee du tableau
            for (int j = i+1; j < a.length; j++)
                if (a[j].compareTo(a[ min ] ) < 0) {
                    min = j;
                }
            // echanger l'element et celui en position i
            Comparable tmp = a[min];
            a[min] = a[i];
            a[i] = tmp;
        }
    }
}
```

Time

```
public class Time implements Comparable {

    private int timeInSeconds;

    public int compareTo(Comparable obj) {

        Time other = (Time) obj;
        int result;
        if (timeInseconds < other.timeInSeconds) {
            result = -1;
        } else if (timeInSeconds == other.timeInSeconds) {
            result = 0;
        } else {
            result = 1;
        }
        return result;
    }
}
```

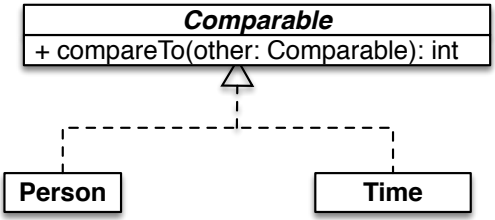
Person

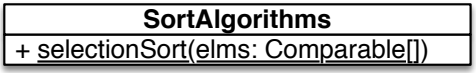
```
public class Person implements Comparable {

    private int id;
    private String name;

    public int compareTo(Comparable obj) {
        Person other = (Person) obj;
        int result;
        if (id < other.id) {
            result = -1;
        } else if (id == other.id) {
            result = 0;
        } else {
            result = 1;
        }
        return result;
    }
}
```

Person (2)

```
public class Person implements Comparable {

    private int id;
    private String name;

    public int compareTo(Comparable obj) {

        Person other = (Person) obj;

        return name.compareTo(other.name);

    }
}
```

UML : Comparable





Fonction de rappel

Problème : uoAlert

- **Problème :** L'Université d'Ottawa souhaite remplacer le système informatique qu'elle utilise pour la **gestion des messages d'alerte**.
  - Le système doit transmettre les messages d'alerte à différents clients, dont une **application sur téléphone portable** et une **application Web**.

uoAlert : AlertListener

- Comme développeur, vous avez l'idée de créer l'interface **AlertListener**.
  - Tout client qui souhaite recevoir les messages d'alerte doit implémenter l'interface **AlertListener**.
  - Le serveur possède une méthode **register** dont le paramètre est de type **AlertListener**.
  - Tout client qui souhaite recevoir les messages d'alerte doit s'inscrire auprès du serveur.

```
public interface AlertListener {
    void processAlert(String message);
}
```

uoAlert : PhoneApp

- L'application **PhoneApp** réalise d'interface **AlertListener**.

```
public class PhoneApp implements AlertListener {

    public void processAlert(String message) {
        System.out.println("PhoneApp: " + message);
    }

    // Les autres methodes de l'application PhoneApp
    // seraient ici.

}
```

uoAlert : WebApp

- L'application **WebApp** réalise d'interface **AlertListener**.

```
public class WebApp implements AlertListener {

    public void processAlert(String message) {
        System.out.println("WebApp: " + message);
    }

    // Les autres methodes de l'application WebApp
    // seraient ici.

}
```

uoAlert : AlertServer

```
public class AlertServer {

    private AlertListener[] clients;
    private int numberOfClients;

    public AlertServer(int capacity) {
        clients = new AlertListener[capacity];
        numberOfClients = 0;
    }

    public void register(AlertListener client) {
        clients[numberOfClients] = client;
        numberOfClients++;
    }

    public void broadcast(String message) {
        for (int i=0; i<numberOfClients; i++) {
            clients[i].processAlert(message);
        }
    }
}
```

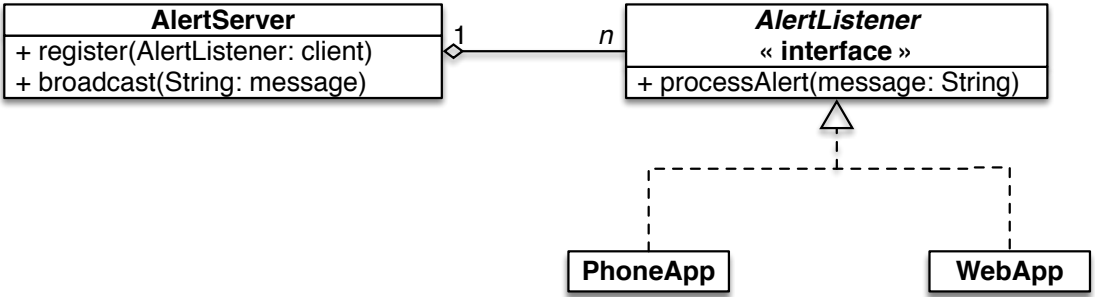
uoAlert : Run

```
public class Run {
    public static void main(String[] args) {
        AlertServer server;
        server = new AlertServer(2);

        PhoneApp phone;
        phone = new PhoneApp();

        WebApp web;
        web = new WebApp();

        server.register(phone);
        server.register(web);

        server.broadcast("Test!");
    }

}
```

uoAlert : Run

```
> java Run
PhoneApp: Test!
WebApp: Test!
```

uoAlert : UML


AlertServer
+ register(AlertListener: client)
+ broadcast(String: message)
1
n
AlertListener
« interface »
+ processAlert(message: String)
PhoneApp
WebApp

Technique de rappel

- La méthode **processAlert** est une **fonction de rappel**.
- Un client s'inscrit (**register**) auprès du service (serveur).
  - Seuls les clients qui possèdent une méthode **processAlert** peuvent s'inscrire.
- Ultérieurement, lorsqu'une alerte survient, le serveur informe tous ses clients. (appel à la méthode **processAlert**)

Prologue

Résumé

- L'**interface** permet la déclaration d'un **type abstrait de données**
- La déclaration d'une **interface** ressemble à la déclaration d'une classe.
  - Sauf, que l'interface ne contient que des méthodes abstraites.
- Le mot clé **implements** nous permet d'indiquer qu'une classe réalise une interface donnée.

Prochain module

- **Héritage**

References I


E. B. Koffman and Wolfgang P. A. T.
Data Structures : Abstraction and Design Using Java.
John Wiley & Sons, 3e edition, 2016.

Marcel **Turcotte**

Marcel.Turcotte@uOttawa.ca

École de **science informatique** et de génie électrique (SIGE)

Université d'Ottawa