LU2IN002 - Introduction à la programmation orientée-objet

Responsable de l'UE : Christophe Marsala
(email: Christophe.Marsala@lip6.fr)

Cours du lundi : Sabrina Tollari
(email: Sabrina.Tollari@lip6.fr)

(support réalisé à partir de ceux de Christophe Marsala et de Vincent Guigue)

SCIENCES SORBONNE UNIVERSITÉ

Cours 8 – lundi 14 novembre 2022

Plan du cours

1. Héritage et interfaces
   - Syntaxe et usage des interfaces
   - Représentation UML des interfaces
   - Propriétés des interfaces
2. Packages

Programme du jour

1. Héritage et interfaces
   - Syntaxe et usage des interfaces
   - Représentation UML des interfaces
   - Propriétés des interfaces
2. Packages
   - Créer un package
   - Compiler et exécuter avec des packages
   - Niveau de visibilité : package

Certaines situations posent problèmes

- Je fais du dessin... Et je veux pouvoir sauver le résultat
- Je fais du dessin... Et je veux afficher facilement le résultat à l'écran

⇒ Limite de l'héritage : pas d'héritage multiple en JAVA

- Rappel : en Java, une classe ne peut hériter que d'une **seule** classe

⇒ Mais un autre outil : **les interfaces** va nous permettre de gérer ce genre de problème

Interface : définition et usage

Usage

Une interface définit :

- un cahier des charges (e.g. `Voiture`, `Circuit`...)
- une propriété (e.g. `Sauvegardable`, `Clonable`...)

Elle donne les fonctions à implémenter et leur signature

Ce que contient une interface

- des signatures de méthodes (comme les méthodes abstraites)
- Mais (<java 1.8) :
  - pas de code
  - pas d'attribut

$\Longrightarrow$ Une interface peut être vue comme une "classe abstraite pure" (classe abstraite sans attribut ni méthode concrète)

Interface : syntaxe

- Déclaration d'une interface `I` contenant une méthode

```
1 public interface I {
2     public void maMethode();  // méthode abstract
3 }
```

- Déclaration d'une classe `A` qui implémente l'interface `I`
  - Comme pour les classes qui héritent d'une classe abstraite, la classe `A` doit définir le code de la méthode abstraite

```
4 public class A implements I {
5     public void maMethode() {
6         // code
7     }
8 }
```

- Une interface ne peut pas être instanciée

```
9 new I(); // ERREUR compilation
```

- Subsomption : un objet de type `A` est aussi de type `I`

```
10 I ia=new A(); // OK
11 ia.maMethode(); // OK
12 ia.methodeDeA(); // ERREUR compil : variable de type I n'est pas un A
13 ((A)ia).methodeDeA(); //OK
```


Interface : exemples d'usage (1)

Un cahier des charges à respecter

★ Vu de l'extérieur de l'objet, que doit faire un véhicule ?

- accélérer, freiner, tourner
- observation (position, direction, vitesse)

```
1  public interface Véhicule {
2    // pour le pilotage
3    public void accelerer(double d);
4    public void freiner(double d);
5    public void tourner(double d);
6
7    // pour l'observation
8    public double getVitesse();
9    public Vecteur getPosition();
10   public Vecteur getDirection();
11 }
```

△ Que des méthodes abstraites (`abstract` n'est pas écrit) ! ! !

- Une classe `Voiture` qui implémente l'interface `Vehicule` doit définir le code de toutes ces méthodes

Interface : exemples d'usage (2)

Les interfaces pour énoncer des propriétés pour des objets

Exemple : la propriété d'être sauvegardable dans un fichier

★ Qu'est ce qu'un objet Sauvegardable ?

Réponse :

- c'est un objet qui peut être sauvegardé sur disque
- c'est un objet qui répond à la méthode suivante :
  - `void save(String filename)`

- On spécifie donc une interface précisant ce comportement :

```
1 public interface Sauvegardable {
2     public void save(String filename);
3 }
```

- Toutes les classes qui veulent avoir cette propriété devront implémenter cette interface

Interface : exemples d'usage (2)

Exemple : la classe Vecteur

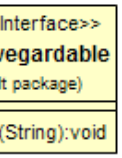
- Un vecteur est un objet sauvegardable
  - ⟹ La classe `Vecteur` doit implémenter l'interface `Sauvegardable`

```
1 public class Vecteur implements Sauvegardable{
2     ...
3     public void save(String filename){
4         ... // instructions à réaliser
5     }
6
7 }
```

- Comme pour les classes abstraites, `Vecteur` doit contenir le code de la méthode (**contrat**) :
  public void save(String filename)
- On est donc sûr de pouvoir sauver un objet `Vecteur`, il y a donc bien une logique de cahier des charges

Interface : représentation UML

- Une interface est représentée en UML par un rectangle (comme les classes)
- Mais en écrivant au dessus du nom de l'interface «interface»
- La relation "une classe implémente une interface" est représentée par une ligne en pointillé avec un triangle au bout



Interface & héritage : propriétés (1/3)

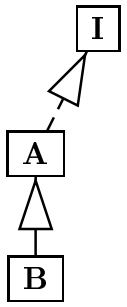
Une classe :

- ne peut hériter que **d'une seule classe**
- mais peut implémenter **plusieurs interfaces**

Exemple : dans un logiciel de géométrie, on peut imaginer plusieurs propriétés : dessinable, déplaçable...

```
1  public interface Dessinable {
2      public void draw();
3  }
4  public interface Deplacable {
5      public void move(double x, double y);
6  }
7  public class Polygone extends Figure implements Dessinable, Deplacable {
8      ...
9      public void draw() {
10         // code
11     }
12     public void move(double x, double y) {
13         // code
14     }
15 }
```


Interface & héritage : propriétés (2/3)



- Une interface peut ne pas contenir de méthodes

```
1 public interface I { }
```

- Si `A` implémente une interface `I` quelconque et `B` hérite de `A` alors :
  - inutile d'écrire dans la signature de `B` que `B` implémente `I`
  - car `B` implémente `I` naturellement (par héritage)

```
2 public class A implements I { }
3 public class B extends A { }
```

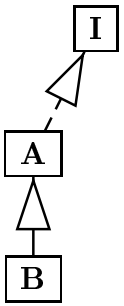
- On peut utiliser `instanceof` pour déterminer si un objet implémente une interface

```
A ab=new B();
if (ab instanceof I) { // true
    ...
}
```

```
String s="Bonjour";
if (s instanceof I) { // false
    ...
}
```

Interface & héritage : propriétés (3/3)

- Si `I` contient la signature d'une méthode que `A` ne peut pas implémenter alors (comme pour classes abstraites) :
  - `A` doit être déclarée abstraite
  - une classe descendante de `A` devra définir le code de la méthode




```
1  public interface I {
2     public void methodeDeI();
3  }
4  public abstract class A implements I { }
5  public class B extends A {
6     public void methodeDeI() {
7        ... // code
8     }
9     ...
10 }
```

- On peut utiliser les méthodes de `Object` à partir d'une variable du type d'une interface

```
11 I ib=new B(); // variable de type I
12
13 ib.toString(); // OK toString de B
14 ib.getClass(); // OK class B
15 ib.methodeDeI(); // OK
16
17 ib.methodeDeB(); // Erreur compil
18 ((B)ib).methodeDeB(); // OK
```


Application

- Il est possible de déclarer une variable d'un type interface
  - Mais jamais d'instanciation d'une interface
  - Seules les méthodes de l'interface (et d'`Object`) sont accessibles
- Application classique : tableau d'un type interface
- Exemple : soit des classes qui implémentent Sauvegardable :
  - classes Vecteur, Point, Figure,...
  - classes Personne, Menagerie,...

```
1 Sauvegardable[] tab = new Sauvegardable[3]; //tableau type interface
2 tab[0] = new Vecteur();
3 tab[1] = new Point();
4 tab[2] = new Menagerie(12);
5 for (int i=0; i<tab.length;i++)
6     tab[i].save("fichier␣"+i);
```

★ Très pratique pour appliquer un traitement identique (ici save) à un ensemble de classes qui n'ont rien à voir entre elles...

★ Une interface peut être implémentée par de nombreuses classes très différentes

Interface qui hérite d'une interface

- Une interface peut hériter d'une autre interface
  - △ C'est un héritage et non pas une implémentation

```
1  public interface Positionnable{
2    public Vecteur getPosition();
3  }
4
5  public interface Deplacable extends Positionnable{
6    public void move(Vecteur v);
7  }
```

- Une classe qui implémente Deplacable :
  - doit fournir une définition pour la fonction move
  - doit aussi fournir une définition pour la fonction getPosition

Une remarque avant de finir sur les interfaces

Hors programme : depuis Java 8, on peut écrire dans une interface :

- des attributs `public static final`
- des méthodes `static`
- des méthodes `default`

Plan du cours

1 Héritage et interfaces

2 Packages
- Créer un package
- Compiler et exécuter avec des packages
- Niveau de visibilité : package

Introduction

Bonne architecture = beaucoup de petites classes...
... chacune étant ciblée, lisible, ré-utilisable
⇒ Le répertoire de projet devient rapidement illisible !

Solution = arborescence de répertoires

- Sous-répertoires associés aux concepts de bas niveaux
- Sous-sous-répertoires de test

Création de packages de classes

Exemple

Gestion d'une course de voitures autonomes

1. Réfléchir à un découpage de bas niveau :
   - **Circuit**
   - **Voiture**
   - Autonome ⇒ gestion de l'**IA** / **stratégies**
2. Ajouter les outils (transverses)
   - Gestion de la **géométrie**
   - Gestion des fichiers (sauvegardes/chargements)
   - Interface graphique (IHM)
3. Package de test :

Idée :

valider le fonctionnement de chaque objet indépendamment du reste du projet (dans la mesure du possible).

⇒ **sous-répertoire de test** dans chaque package principal

Package

Package java

- Un package est un ensemble de classes mises dans un même répertoire

- Pour définir un package

```
package nomdupackage; // au début du fichier
```

- Pour importer **une classe** d'un package

```
import nomdupackage.MaClasse;
```

- Pour importer **toutes les classes** d'un package

```
import nomdupackage.*;
```

Convention d'écritures

Les noms des packages s'écrivent tout en minuscules
Exemples : `java.lang`, `java.util`, ...

Créer un package

Package java

- **Règle java** : nom du répertoire = nom du package
- Chaque classe précise le package dans lequel elle est

Exemple : on veut définir un package `paquet1` avec 2 classes `A` et `B`

- Contenu du fichier `A.java` (situé dans le répertoire `paquet1`)

```
1 package paquet1;
2
3 public class A {
4     ...
5 }
```

- Contenu du fichier `B.java` (situé dans le répertoire `paquet1`)

```
1 package paquet1;
2
3 public class B {
4     ...
5 }
```

△ Une classe ne peut appartenir qu'à un seul package

Créer un sous-package

Exemple (suite) : on veut créer une classe `TestA` dans un sous-package `souspaquet1` pour tester la classe `A`

- dans le fichier `TestA.java` du répertoire `souspaquet1`

```java
package paquet1.souspaquet1; // création du sous-package

import paquet1.A; // importation pour pouvoir utiliser A

public class TestA {
    public static void main(String [] args) {
        A a1=new A();
    }
}
```

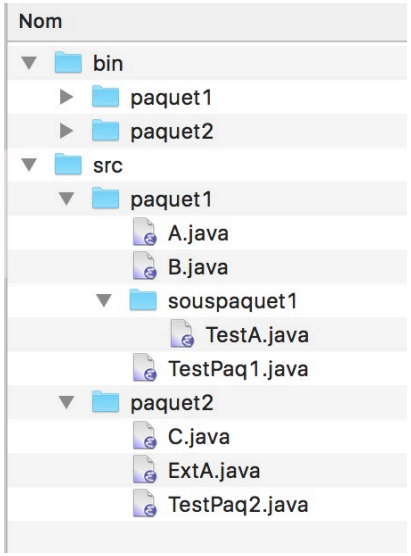
△ Le répertoire `souspaquet1` se trouve dans le répertoire `paquet1`

△ Le nom du package est `paquet1.souspaquet1`

△ Le nom complet de la classe est `paquet1.souspaquet1.TestA`

Déclarations obligatoires

Arborescence :



1 Déclaration de paquet

```java
// Fichier A.java
package paquet1;
public class A {
...
```

2 Sous-package

```java
package paquet1.souspaquet1;
import paquet1.A;
public class TestA {
    public static void main(String[] args) {
        // tests specifiques a A
```

3 Autre package

```java
package paquet2;
import paquet1.A;
public class ExtA extends A { // classe fille
    public ExtA() {
        super();
    }
```

4 Classe JDK

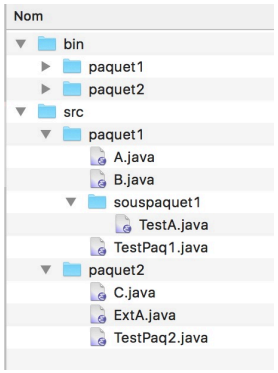
```java
import java.util.ArrayList;
```


Compilation / exécution du code

- **Compilation** (position = racine)
  - Spécification d'un répertoire cible : `-d`
  - Spécification du répertoire de gestion des sources : `-cp`

```
» javac -cp src -d bin src/paquet1/souspaquet1/TestA.java
```

⟹ Compile l'exécutable + toutes les dépendances (ici `A` mais pas `B` ni les autres classes)



- **Exécution**
  - Instruction pour se positionner dans le répertoire d'exécution : `-cp`
  - Chemin avec des `.` (pas des `/`)

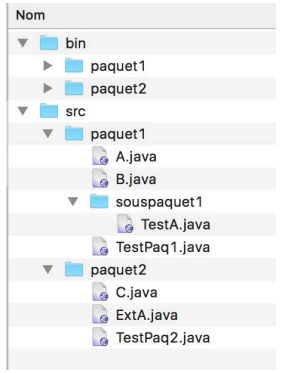
```
» java -cp bin paquet1.souspaquet1.TestA
```


Niveaux de visibilité : package

introduction des packages = subtilités sur la visibilité

```java
package paquet1;
public class A {
  public int i;     // public
  protected int j; // protected
  private int k;    // private
  int n;            // package (nouveau)

  public A(){
      i=1; j=2; k=3; n=4;
  }
}
```



Visibilités des attributs de `A` depuis :

		i	j	k	n
Même package	`B, TestPaq1`	√	√	×	√
Classe fille	`ExtA`	√	√	×	×
Autres cas	`C,TestPaq2,TestA`	√	×	×	×