Cours Programmation Orientée Objet en Java

Cours 3

Anne Parrain

UFR des Sciences - Université d'Artois
Licence Informatique
Semestre 4

Année universitaire 2018–2019

Plan du cours

1. La notion de package

2. Les interfaces

3. Les chaînes de caractères

Section 1

La notion de package

Organisation générale d'un projet Java

Pour le moment, nos applications `Java` sont constituées de

- un ensemble de classes
- chaque classe est dans un fichier
- le nom du fichier est *exactement identique* au nom de la classe, aux majuscules près.

Organisation générale d'un projet Java

Dans les projets plus grands :

- les classes peuvent être rangées dans différents répertoires et sous-répertoires
- les classes qui sont dans un même répertoire sont dans un même paquetage ou `package`
- le nom d'un `package` est composé des noms des répertoires formant le chemin d'accès à ses classes, séparés par un `.`
- les noms des répertoires commencent par une minuscule.

Déclaration de package

- La définition des classes commence par la désignation du package dans lequel la classe se trouve.
- Cette indication est *optionnelle*.
- Si cette instruction est présente, elle doit être la première instruction du fichier.

```
package etudiant;

class Etudiant{...
}
```

Déclaration de package

Les très gros projets sont plus généralement organisés en une hiérarchie de répertoires à plusieurs niveaux.

```
package monrep.monsousrep;
```

Le chemin `monrep.monsousrep` doit indiquer la liste des sous-répertoires à traverser pour atteindre cette classe.

Les API Java

`API` : *Application Programming Interface*

C'est l'ensemble des classes Java déjà programmées auxquelles vous avez accès.

Elles sont organisées en packages et sous-packages. En voici quelques uns :

- `java.lang`
- `java.util`
- `java.awt`
- `java.awt.event`
- `javax.swing`
- ...

Importation de classes

- Par défaut, accès à toutes les classes définies :
  - dans `java.lang`
  - dans votre répertoire courant
- Pour utiliser des classes provenant d'autres `package`s, on utilise l'instruction `import`.
- Par exemple

  ```
  import java.util.Date;
  ```
- Plus généralement

```
import rep1.rep2.UneClasse;
import rep1.rep3.*;
```

- Les instructions `import` doivent se trouver après l'instruction `package`, si elle est présente, et avant la définition de la classe.

Utiliser les classes d'un package

Pour pouvoir utiliser une classe `MaClasse` d'un package `monPackage`, il faut :

- que `MaClasse` soit définie `public` dans le package `monPackage` ;
- que le code qui veut utiliser `MaClasse` l'indique par une instruction
  `import monPackage.MaClasse;`
  ou bien
  `import monPackage.*;`.

Visibilité d'une classe

- Dans un même package, les classes sont définies `public` ou rien.
- Si une classe est déclarée `public`, alors elle est accessible en dehors du package auquel elle appartient.
- Par défaut, (i.e. sans `public`), l'accès à la classe sera dit *package* et la classe ne sera accessible que par les classes du même package.

La variable `CLASSPATH`

La variable `classpath` permet au compilateur `javac` et à l'interprète `java` de savoir où se trouvent les classes utilisées.

Par défaut, `classpath` contient `.` et le chemin d'accès aux API Java.

Organisation des fichiers sources, bytecode et documentation

- les fichiers `.java` organisés en répertoire selon l'architecture des packages sont *sous* le répertoire `src`
- une même architecture est reproduite par la compilation (fichiers bytecode `.class`) sous le répertoire `bin`
- la documentation est toujours dans le répertoire `doc`

Section 2

Les interfaces

Les interfaces

Décomposer un problème en sous-problèmes :

- c'est définir le contrat à respecter par chaque partie
- c'est définir les protocoles de communication entre objets

contrat ≡ interface

Une interface n'est pas une classe :

- elle ne peut pas être instanciée (pas de `new`) ;
- elle n'implémente pas la plupart de ses méthodes.

Une interface ne peut contenir que

- la signature de méthodes (`public`)
- la déclaration de constantes (`public static final`)
- un comportement par défaut pour certaines méthodes

Exemple

```
public interface Article {
  public String getDescriptif();
  public int getQuantite();
  public float getPrix();
  public float montantStock();
  public Article ajouteQuantite(int nb);
  public Article retireQuantite(int nb);
}
```

Déclaration d'une interface

- Une interface est dans un fichier du même nom que l'interface ;
- Une interface peut être déclarée dans un package ;
- Une interface est `public` ou rien (i.e. *package*) ;
- Un fichier qui contient une définition d'interface peut contenir des instructions `import` ;

Implémentation d'une interface

Une classe, si elle implémente une interface, doit fournir une implémentation pour **toutes** les méthodes définies dans l'interface.

Une classe peut implémenter plusieurs interfaces.

Syntaxiquement :

```
class NomDeLaClasse implements NomDUneInterface,
NomDUneAutreInterface{ ...}
```

Exemple

```
public class Crayon implements Article{
   private String nom;
   private String couleur;
   private int qte;
   private float prix;
   public Crayon(String nom, String couleur, float prix){
     this.nom = nom;
     this.couleur = couleur;
     this.prix = prix;
   }
   public Crayon(String nom, String couleur,
                 float prix, int qte){
     this(nom, couleur, prix);
     if (qte > 0)
         this.qte = qte; }
```

Exemple

```
public int getQuantite(){ return qte; }
public float getPrix(){ return prix; }
public String getDescriptif( ){
  return nom+" - "+couleur;  }
public String getNom(){ return nom; }
public String getCouleur(){ return couleur; }
public float montantStock(){
    return prix*qte;
}
```

Exemple

```
  public Crayon ajouteQuantite(int nb){
    if (nb > 0)
      return new Crayon(nom,couleur,prix,qte+nb);
    return null;
  }
  public Crayon retireQuantite(int nb){
    if (nb < 0)
      return null;
    if (qte >= nb)
      return new Crayon(nom,couleur,prix, qte - nb);
    return new Crayon(nom,couleur,prix); }
}//Crayon
```

Exemple

```
public class Vetement implements Article{
   private String type;
   private String couleur;
   private int taille;
   private int qte;
   private float prix;
   public Vetement(String type, String couleur,
                   int taille, float prix){
     this.type = type;
     this.taille = taille;
     this.couleur = couleur;
     this.prix = prix; }
   public Vetement(String type, String couleur, int taille,
                   float prix, int qte){
     this(type, couleur, taille, prix);
     if (qte > 0)
         this.qte = qte; }
```

Exemple

```
public int getQuantite(){
  return qte; }
public float getPrix(){
  return prix; }
public String getType( ){
  return type; }
public String getCouleur(){
  return couleur; }
public int getTaille(){
  return taille; }
public String getDescriptif( ){
  return nom+" - "+couleur+" - taille : "+taille; }
```

Exemple

```
  public Vetement ajouteQuantite(int nb){
    if (nb > 0)
      qte += nb;
    return this;
  }

  public Vetement retireQuantite(int nb){
    if (nb > 0){
      if (qte >= nb)
        qte = qte - nb;
      else qte = 0;
    return this;
  }

  public float montantStock(){
       return prix*qte;
  }
}//Vetement
```

Typage

- Les définitions d'interface créent des noms de types tout comme le font les définitions de classe.
- On peut utiliser le nom d'une interface comme le nom de type d'une variable ;
- Toute instance d'une classe qui implémente cette interface peut être affectée à cette variable.

Typage

```
Article a1 = new Vetement("robe","jaune",38,55);
Article a2 = new Crayon("crayon d'aquarelle","rouge",3);
Vetement v = new Vetement("pantalon","gris",44,40);
Vetement v2 = a1.ajouteQuantite(2); // Refus
Vetement v3 = v.ajouteQuantite(1);
System.out.println("Taille : "+v.getTaille());
System.out.println("Taille : "+a1.getTaille()); // Refus
```

Conflit de noms

Une classe peut implémenter plusieurs interfaces.
Que se passe-t-il si un même nom de méthode se trouve défini dans plusieurs interfaces ?
Plusieurs cas :

- même signature. *Pas de conflit*.
- les signatures différent par les paramètres.
  Pas de conflit : surcharge de méthodes
- les signatures ne diffèrent que par leur type de retour.
  Conflit ! Il n'est pas possible qu'une classe implémente ces deux interfaces en même temps.

L'interface Comparable

L'interface Comparable est une interface du package java.lang

```
public interface Comparable<T> {
  int compareTo(T o);
}
```

L'interface Comparable (2)

La classe Crayon peut implémenter ces interfaces :

```
public class Crayon implements Article, Comparable<Crayon> {
    public int compareTo(Crayon a){
        if (this.nom.compareTo(a.getDescriptif()) < 0)
            return -1;
        else {
            if (this.nom.compareTo(a.getDescriptif()) == 0){
                if (this.prix < a.getPrix())
                    return -1;
                else {
                    if (this.prix == a.getPrix()){
                        if (this.qte < a.getQuantite()) return -1;
                        else return (this.qte ==
                            a.getQuantite()?0:1);
                    } else return 1;
                }
            } else return 1;
        }
    }
```

Extension d'interface

Une interface peut étendre une interface déjà existante :

```
public interface Article extends Comparable<Article> {

    public String getDescriptif();
    public int getQuantite();
    public float getPrix();
    public float montantStock();
    public Article ajouteQuantite(int nb);
    public Article retireQuantite(int nb);
}
```

Les classes qui implémentent `Article` doivent également fournir une implémentation pour

```
    public int compareTo(Article a);
```

Extension d'interface (2)

On devra trouver dans la classe `Crayon` maintenant :

```
public class Crayon implements Article {
    public int compareTo(Article a){
        if (this.nom.compareTo(a.getDescriptif()) < 0)
            return -1;
        else {
            if (this.nom.compareTo(a.getDescriptif()) == 0){
                if (this.prix < a.getPrix())
                    return -1;
                else {
                    if (this.prix == a.getPrix()){
                        if (this.qte < a.getQuantite()) return -1;
                        else return (this.qte ==
                            a.getQuantite()?0:1);
                    } else return 1;
                }
            } else return 1;
        }
    }
```

Comportement par défaut

Les interfaces peuvent proposer un comportement *par défaut* pour certaines de ses méthodes grâce au mot-clé `default`
On propose une nouvelle définition de l'interface `Article`

```
public interface Article {
  public String getDescriptif();
  public int getQuantite();
  public float getPrix();
  public Article ajouteQuantite(int nb);
  public Article retireQuantite(int nb);
  default public float montantStock(){
    return getPrix()*getQuantite();
  }
}
```

Maintenant `Crayon` et `Vetement` possèdent une implémentation de la méthode `montantStock()` et n'ont pas besoin de la définir.

Comportement par défaut (2)

Attention ! On n'aurait pas pu écrire :

```
public interface Article {
  public String getDescriptif();
  public int getQuantite();
  public float getPrix();
  public Article ajouteQuantite(int nb);
  public Article retireQuantite(int nb);
  default public float montantStock(){
    return this.prix * this.qte;
    // Erreur ! on ne peut rien supposer sur les classes
    // qui implementeront Article que les methodes definies
        dans Article
  }

}
```

Comportement par défaut (3)

Pour reprendre notre exemple précédent, on pourrait proposer :

```
public interface Article extends Comparable<Article> {
    public String getDescriptif();
    public int getQuantite();
    public float getPrix();
    default public float montantStock(){
        return getPrix()*getQuantite();
    }
    public Article ajouteQuantite(int nb);
    public Article retireQuantite(int nb);
    default public int compareTo(Article a){
        if (getDescriptif().equals(a.getDescriptif())) {
            if (getQuantite()>a.getQuantite())
                return 1;
            else return (getQuantite()==a.getQuantite()?0:-1);
        }
        else return getDescriptif().compareTo(a.getDescriptif());
    }
}
```

Comportement par défaut (4)

- `Crayon` et `Vetement` ne sont pas obligées de redéfinir `compareTo` $\Longrightarrow$ elles utiliseront alors le comportement par défaut défini dans `Article`
- mais elles peuvent redéfinir ce comportement s'il ne correspond pas à leur besoin. `Crayon` peut continuer de définir sa méthode `compareTo` comme précédemment

Comportement par défaut (5)

- elles peuvent aussi réutiliser ce comportement par défaut partiellement (ou complètement) par son appel explicite `NomInterface.super.appelMethode()`

```
public class Vetement implements Article {
    public int compareTo(Article a){
        int res = Article.super.compareTo(a);
        // en cas d'egalite sur le descriptif et la qte,
        // on classe par rapport au prix
        if (res == 0) {
            if (this.prix < a.getPrix())
                return -1;
            else return (this.prix == a.getPrix()?0:1);
        }
        return res;
    }
```

Conflit d'implémentation par défaut

Puisqu'une classe peut implémenter plusieurs interfaces.

Que se passe-t-il si un même nom de méthode se trouve défini dans plusieurs interfaces ?

Plusieurs cas :

- même signature
  - Pas de comportement par défaut défini ou une seule interface fournit un comportement par défaut
    Pas de conflit.
  - Plusieurs interfaces définissent un comportement par défaut pour cette méthode
    Conflit !
    - La classe peut lever l'ambigüité en redéfinissant cette méthode
      Plus de conflit.
- les signatures différent par les paramètres.
  Pas de conflit : surcharge de méthodes
- les signatures ne diffèrent que par leur type de retour.
  Conflit ! Il n'est pas possible qu'une classe implémente ces deux interfaces en même temps.

Section 3

Les chaînes de caractères

La classe String

- à noter : pas obligatoirement de `new` pour créer un objet String ;
- une `String` est implémentée comme un tableau **constant** de caractères ;
- $\Rightarrow$ on ne modifie pas une String, on créé un nouvel objet String (la classe String est une classe non modifiable).

La classe String

Quelques méthodes :

```
int length();
char charAt(int ind);
boolean equals(String s);
int indexOf(char c);
String substring(int debut,int fin);
String substring(int debut);
String trim();
```

La classe StringBuffer

La classe `StringBuffer`

- implémente les chaînes de caractères, comme la classe `String`
- à la différence qu'une instance de `StringBuffer` est modifiable.

$\Rightarrow$ plus efficace lorsqu'on effectue beaucoup de modifications sur une chaîne de caractères.

La classe StringBuffer (2)

- Les principales méthodes :
  - `append`,
  - `insert`,
  - `setCharAt`,
  - ...

Exemple

On veut qu'une promotion affiche tous ses étudiants :
On pourrait écrire

```
  // Dans la classe Promotion
  public String toString(){
    String tmp = "Promotion : " + nom;
    if (nbEtudiants > 0) {
      tmp += "Liste des etudiants :\n");
      for (int i = 0; i < nbEtudiants; i++){
        tmp += lesEtudiants[i].toString() + "\n";
      }
    }
    else
      tmp += "\n Pas d'etudiant pour le moment!";
    return tmp;
}
```

Exemple

Mais il faut écrire :

```
  // Dans la classe Promotion
  public String toString(){
    StringBuffer sb = new StringBuffer();
    sb.append("Promotion : ");
    sb.append(nom);
    if (nbEtudiants > 0) {
      sb.append("\n Liste des etudiants :\n");
      for (int i = 0; i < nbEtudiants; i++){
        sb.append(lesEtudiants[i].toString());
        sb.append("\n");
      }
    }
    else
      sb.append("\n Pas d'etudiant pour le moment !");
    return sb.toString();
}
```