Séquence 10

Paquetages et accesibilité des composants

1 Introduction

Les paquetages sont une façon de mettre de l'ordre dans les programmes java en répartissant les classes en plusieurs sous-ensembles. Un paquetage (package en anglais) est comme un dossier qui peut contenir plusieurs classes et des sous-dossiers (paquetages imbriqués).

On peut désigner une classe au moyen d'un chemin d'accès qui décrit dans quels paquetages il faut la chercher. Nous connaissons par exemple la classe Date. Son chemin d'accès est `java.util.Date`. En partant de la racine des classes Java, il faut d'abord aller dans le paquetage `java`, puis dans le sous-paquetage `util` et là, il y a la classe Date.

Ce nom complet se voit dans la clause `import` que l'on utilise en début de programme pour pouvoir utiliser la classe.

Un programme peut être écrit dans un paquetage. C'est souvent le cas lorsqu'on utilise Eclipse : créer un projet crée un paquetage dans lequel on met les classes du programme. Pour les gros programme, il est possible de diviser le programme en plusieurs paquetages ou en sous-paquetages du paquetage principal du projet. Le but de la définition de plusieurs paquetages est de séparer le code en plusieurs modules relativement indépendants les uns des autres.

2 Définir le paquetage d'une classe

Le nom du paquetage auquel appartient une classe doit apparaître en début de programme. Ce doit être la première ligne de code du fichier (il peut y avoir des commentaires ou lignes blanches avant).

Ainsi le code de la classe Date doit commencer par la déclaration :

```
package java.util;
```

Il n'y a pas de déclaration spécifique du paquetage qui serait indépendante des classes qu'il contient. Un paquetage existe à partir du moment où il apparaît dans un chemin d'accès à une classe.

Sur l'ordinateur, les paquetages doivent correspondre à des dossiers. Supposons que je veuille créer *à la main* une classe avec le chemin d'accès `paquetage.sousPaquetage.ClasseA`. Il faut que je crée un dossier `paquetage`, un sous-dossier `sousPaquetage` dans le dossier `paquetage` et que je mette le code source (fichier .java) dans ce sous-paquetage.

Pour compiler et exécuter la classe dans une fenêtre de commande, il faut que je soit non pas dans le dossier de la classe ni dans le dossier `paquetage` mais dans le dossier père de `paquetage`. Et

là, pour compiler, je dois donner le chemin d'accès au fichier source (sous forme d'un chemin d'accès à un fichier). Pour exécuter le programme, je dois donner le chemin d'accès en java à la classe.

```
> javac paquetage\sousPaquetage\ClasseA.java
> java paquetage.sousPaquetage.ClasseA
```

Dans un environnement de programmation comme Eclipse, le système gère le système de dossier. Par exemple, si l'on crée un sous-paquetage avec l'option `file -> new -> package` (dans le menu *file*, option *new* et sous-option *package*) , cela crée un nouveau sous-dossier dans `src`.

3 Accès à une classe dans le code Java

Dans le code d'une méthode ou la déclaration d'attribut d'une classe, on peut accéder directement aux composantes d'une autre classe en utilisant simplement le nom de cette autre classe dans deux cas :

— les deux classes sont dans le même paquetage
— la classe utilisée appartient au paquetage `java.lang` qui contient les classes les plus essentielles du langage comme par exemple `String`, `Math`, `System` et `RuntimeException`.

Dans les autres cas, on ne peut pas accéder à la classe directement. Il faut soit utiliser le chemin d'accès complet depuis la racine, soit écrire une clause `import`.

```
public class UneDate{
    public static void main(String[] args){
        java.util.Date ladate;
        ladate = new java.util.Date();
        System.out.println(ladate.toString());
    }
}
```

Pour éviter d'avoir à utiliser le nom complet, on peut importer une classe ou l'ensemble des composantes d'un paquetage. Le code suivant importe la seule classe Date.

```
import java.util.Date;
public class UneDateBis{
    public static void main(String[] args){
        Date ladate;
        ladate = new Date();
        System.out.println(ladate.toString());
    }
}
```

Pour importer toutes les classes du paquetage `java.util` (notamment `Date` et `ArrayList`), il faut écrire :

```
import java.util.*;
```

4 Les paquetages comme unité d'encapsulation

La notion de paquetage est utilisée pour définir l'accessibilité d'un composant logiciel (classe, attribut ou méthode).

Il existe quatre modes différents utilisant les trois mots-clés `public`, `private`, `protected`.
Les règles sont les suivantes :

— Un composant défini avec `public` est accessible depuis toute autre classe Java sans restriction.
— Un composant défini avec `private` n'est accessible qu'à l'intérieur de la classe où il est défini. Notons qu'une classe ne peut pas être déclarée avec ce mode : il ne peut être utilisée que pour les composants d'une classe.
— Un composant défini avec `protected` est accessible dans les autres classes du même paquetage et également dans les sous-classes de la classe actuelle. Ces sous-classes peuvent appartenir à d'autres paquetages.
— Un composant défini avec aucun des trois mots-clés est accessible par les autres classes du même paquetage.

Comme on le voit, les différents modes ne couvrent pas toutes les possibilités. Il n'y a pas de mode pour qu'une variable ou une méthode soit accessible dans les sous-classes mais pas dans le paquetage.

Si l'on essaie d'accéder à une méthode ou une variable hors de son domaine de visibilité, on a un message d'erreur comme suit.

```
error: meth1() has private access in Class1A
error: meth2() has protected access in Class1A
error: meth3() is not public in Class1A; cannot be accessed
                                        from outside package
```

5 Structuration de la librairie Java

La librairie standard Java, celle qui est présente sur toute plateforme d'exécution Java, utilise une hiérarchie de paquetages pour ordonner ses milliers de classes. Un premier niveau de structuration distingue un petit nombre de paquetages :

— le paquetage `java` contient toutes les classes essentielles du langage.
— le paquetage `java` contient des classes moins essentielles, notamment des classes servant pour l'interface graphique ou l'utilisation du langage de représentation XML.
— le paquetage `org` contient des classes en rapport avec des normes édictées par des organismes non commerciaux (par exemple les normes du web édictées par le consortium W3C).
— dans certaines versions de Java, il y a aussi un paquetage `com` pour des classes liées à des entreprises commerciales, en l'occurrence, l'entreprise propriétaire du langage Java.

Voyons quelques sous-paquetages représentatifs :

— `java.awt` contient des classes d'interface graphique (par exemple des classes pour les fenêtres, les boutons, les actions de la souris, etc).
— `java.io` contient des classes permettant de gérer des fichiers. Le terme `io` est une abbréviation pour *input-output* (entrées-sorties).
— `java.lang` contient les classes essentielles du langage (`Math`, `Exception`, `System`, `String`, etc). On accède directement à ces classes sans importation du paquetage.
— `java.net` contient les classes liées au réseau.
— `java.sql` contient les classes d'interface avec les bases de données.
— `java.util` contient divers utilitaires dont les dates et les structures de données (`ArrayList`, `Vector`, etc).
— `javax.swing` contient des composants d'interface graphique plus puissants et plus faciles à utiliser que ceux de `java.awt`.

Il existe de nombreux autres sous-paquetages. De plus, les sous-paquetages mentionnés possèdent des sous-sous-paquetages. Il peut y avoir jusqu'à 5 niveaux d'imbrication de paquetage (un chemin complet d'accès à une classe peut avoir jusqu'à 5 points).

6 Protection des données

Il est d'usage de protéger les données au moyen d'un mode associé aux attributs. Le mode `private` offre un haut degré de protection mais il est trop restrictif lorsqu'on veut utiliser l'héritage. On a alors recours au mode protected qui ouvre l'accès à l'attributs non seulement aux sous-classes mais aux autres classes du paquetage.

Protéger les attributs avec un mode est efficace lorsqu'il s'agit d'attributs de types primitifs comme int ou boolean. Dans le cas des types références, il ne suffit pas de protéger l'attribut : il faut également ne pas révéler la référence. Voyons un exemple où l'attribut contient un tableau.

6.1 Exemple d'un problème lié à un accesseur

Prenons l'exemple d'un parking qui permet de garer un certain nombre de voitures. Quand une voiture arrive, une méthode permet d'enregistrer son immatriculation dans un tableau et quand une voiture part, une méthode permet de retirer cette immatriculation du tableau. Cette immatriculation est une chaîne de caractères et les cases vides contiennent la valeur `null`. Le tableau sera stocké dans une variable `private` pour la protéger et un accesseur permettra d'accéder au tableau.

```
public class Parking {
    private String[] park;

    public Parking(int nbPlaces){
        park = new String[nbPlaces];
    }

    public String[] getContent(){
        return park;
    }

    public void arriver(String num){
        int i=0;
        while(i<park.length && park[i]!=null)
            i++;
        if (i==park.length)
            throw new PlusDePlaceException();
        else
            park[i]=num;
    }

    public void partir(String num){
        int i = 0;
        while(i<park.length && ! park[i].equals(num))
            i++;
        if (i==park.length)
            throw new java.util.NoSuchElementException();
        else
```

```
            park[i]=null;
    }

    public String toString(){
        String res = "";
        int nbvides=0;
        for (String str: park)
            if (str==null)
                nbvides++;
            else
                res = res + str + ",␣";
        res = res + nbvides + "␣places␣vides";
        return res;
    }

}
```

Pour la cohérence de nos objets, on veut que les voitures ne puissent venir que via la méthode `arriver` et ne partir que via la méthode `partir`. Le code suivant permet de constater que ces propriétés ne sont pas assurées par notre code.

```
public class ParkingMain {
    public static void main(String[] args) {
        Parking sigare = new Parking(5);
        sigare.arriver("aa-045-aa");
        sigare.arriver("aa-047-aa");
        sigare.arriver("bb-047-cc");
        System.out.println(sigare);
        String[] tab = sigare.getContent();
        tab[0] = "zz-999-xx";
        tab[1] = null;
        System.out.println(sigare);
    }
}
```

L'exécution de ce programme donne l'affichage suivant.

```
> java ParkingMain
aa-045-aa, aa-047-aa, bb-047-cc, 2 places vides
zz-999-xx, bb-047-cc, 3 places vides
```

Les deux lignes correspondent à deux états successifs de l'objet appelé `sigare`. On voit dans cet exemple d'exécution que les voitures aa-045-aa et aa-047-aa ont quitté le parking sans appel à `partir` et la voiture zz-999-xx est arrivée sans appel à `arriver`.

La méthode `getContent` renvoie un tableau. Plus précisément, elle renvoie l'adresse de ce tableau dans le tas. Une fois cette adresse renvoyée, elle est mise dans une variable du `main`. On voit sur cet exemple très clairement que le mot-clé `private` s'applique à la variable `park` et non pas au tableau lui-même. Cette protection est efficace dans la mesure où la variable `park` n'est pas utilisable dans la `main` (parce qu'il est dans une autre classe que `Parking`) mais que cette protection n'assure pas l'intégrité des données à cause de l'accesseur `getContent` qui renvoie l'adresse du tableau.

6.2 Solution : copier le tableau

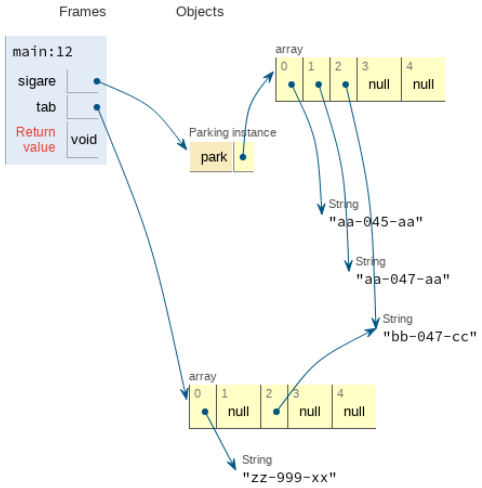
La solution à ce problème consiste à ne pas renvoyer le tableau utilisé dans l'objet mais à en faire une copie qui est renvoyée.

```
public String[] getContent(){
    String[] copieDePark = new String[park.length];
    for (int i=0; i<park.length; i++)
        copieDePark[i] = park[i];
    return copieDePark;
}
```

Avec ce nouveau code de la méthode `getContent`, lorsqu'on exécute le programme `ParkingMain`, on constate que les données ne sont pas perturbées par les affectations de la variable `tab`. Les deux variables `park` et `tab` désignent deux tableaux différents sans interactions entre les deux.

```
> java ParkingMain
aa-045-aa, aa-047-aa, bb-047-cc, 2 places vides
aa-045-aa, aa-047-aa, bb-047-cc, 2 places vides
```

Voici l'état de la mémoire à la fin de l'exécution du programme : on voit bien les deux tableaux différents.



6.3 Et dans le cas d'un objet

Lorsque l'attribut à protéger est un objet, il y a le même risque d'atteinte à l'intégrité des données si une méthode renvoie la référence de l'objet. Par exemple, si dans une classe `Personne`, il y a une méthode `getDateNaissance` qui renvoie une instance de `Date`, il est possible de changer la date de naissance simplement en utilisant un modificateur de la classe `Date` comme par exemple `setYear`.

```
lapersonne.getDateNaissance().setYear(0);
```

Ce simple code a changé la date de naissance de la personne ! Ce qui n'est pas souhaitable. Deux solutions : comme dans le cas des tableaux, la méthode `getDateNaissance` peut renvoyer une copie de la date de naissance et non pas l'objet contenu dans l'attribut.

Dans certains cas, on ne voudra pas faire une copie de l'objet parce que c'est trop coûteux (cas d'un objet complexe ou qui occuppe beaucoup de place en mémoire). Dans ce cas, il vaut mieux renoncer à la méthode `getDateNaissance` et lui substituer des méthodes qui renvoient une information partielle (par exemple des méthodes qui renvoient l'année, le mois et le jour de naissance, donc des int et non des objets).

6.4 Conclusion sur les structures internes à une classe

Comme l'illustre l'exemple du parking, il faut se demander si les données d'une classes doivent être dévoilées à l'extérieur de la classe au moyen de méthodes de type accesseur. La question se pose pour les tableaux, mais aussi pour certains objets comme par exemple les `ArrayList` qui jouent un rôle analogue aux tableaux.

Il n'y a pas de réponse générale : il s'agit d'un élément important qui dépend de l'application, des interactions entre objets, des contraintes. Cela fait partie des choix à opérer lors de la conception d'un programme.