Paquetages (packages an anglais)

Un paquetage est une collection de classes et d'interfaces. Un paquetage regroupe des classes ayant des liens entre elles, parce qu'elles travaillent sur un même domaine, qu'elles concernent un même sujet…

Une classe se trouve dans le paquetage `nom_paquetage` si la première ligne du fichier source de la classe contient l'annonce :

```
package nom_paquetage;
```

Le nom d'un paquetage est une suite d'identificateurs séparés par des points, comme par exemple `java.util`. Dans le système de fichiers, un paquetage correspond à un répertoire, par exemple, le paquetage de nom `projet.location` doit être placé dans un répertoire `projet/location/` (ou `projet\location\` sous Dos). Les points sont remplacés par un slash (ou anti-slash) au niveau de l'arborescence. Tous les fichiers de bytecode (fichiers .class) du paquetage `projet.location` doivent se trouver dans le répertoire `projet/location/`. Pour cela, on peut, lors de la compilation, utiliser l'option `-d` qui permet de spécifier le répertoire dans lequel seront stockés les fichiers .class.

```
javac -d repertoire file.java
```

Remarque : si un fichier source ne contient pas de déclaration **`package`** `…;`, les classes sont dans un paquetage par défaut (paquetage sans nom) qui correspond au répertoire courant.

Au sein d'un paquetage on a accès aux classes de ce paquetage. Les classes des autres paquetages ne sont accessibles que si elles sont déclarées publiques (**`public`**). Une classe qui n'est pas déclarée publique n'est donc accessible qu'au sein de son paquetage.

Une classe nommée `C` qui se trouve au sein du paquetage p a pour nom complet `p.C`. L'accès à cette classe depuis un autre paquetage se fait par son nom complet ou en utilisant la directive import en tête du fichier source : **`import`** `p.C;` ou **`import`** `p.*;`

API Java

La bibliothèque standard de classes Java 6[1] est accessible en ligne sur :

https://docs.oracle.com/javase/7/docs/api/

Elle est organisée en paquetages thématiques :

- `java.lang` : classes de base du langage (chaînes, maths, processus, exceptions…)
- `java.util` : structures de données (vecteurs, piles, tables, parcours…)
- `java.io` : entrées sorties classiques (texte sur clavier-écran, fichiers…)
- `java.awt` : interfaces graphique (fenêtrage, événements…)
- `java.net` : communications Internet (manipulation d'URL, de sockets…)
- `java.applet` : insertion de programmes dans des documents HTML
- …

Le paquetage `java.lang`

Ce paquetage contient la classe **`Object`** et la plupart des autres classes courantes de Java (**`String`**, **`Exception`**…). Il contient aussi les classes enveloppes (wrappers) associées aux types simples : classe **`Double`** associée à **`double`**, classe **`Integer`** associée à **`int`**…

Égalité d'objets

On rappelle que tous les objets (et tableaux) en Java sont manipulés par références. Utilisé sur des références, l'opérateur == compare les valeurs de références. Par exemple, si `s1` et `s2` sont deux chaînes de caractères (deux références sur des **`String`**), **`if`** `(s1 == s2)` comparera les références (la condition est vraie ssi `s1` et `s2` repèrent le même objet). Or, il est souvent nécessaire de comparer les contenus des objets.

Pour savoir si deux objets d'une classe `A` ont des contenus identiques, on pourra définir dans `A` sa propre méthode d'égalité mais il est mieux de redéfinir la méthode `equals()` héritée de **`Object`**.

Par exemple, la classe **`String`** redéfinit la méthode `equals()` et on trouve dans la documentation de la classe **`String`** :

[1] Différentes versions de Java se sont succédées qui ont chacune apporté leurs nouveautés. La version actuelle est la version 7. La documentation de l'API Java précise depuis quand chacune des fonctionnalités a été ajoutée.

equals

public boolean equals(Object anObject)
Compares this string to the specified object. The result is true if and only if the argument is not null and is a String object that represents the same sequence of characters as this object.
Overrides:
equals in class Object
Parameters:
anObject - the object to compare this String against.
Returns:
true if the String are equal; false otherwise.

Une méthode `equals(Object)` convenablement codée est généralement de la forme :

```
public class A {
     …
     public boolean equals(Object o) {
          if (this == o)  // si l'objet o est l'object courant,
               return true;    // alors ils sont égaux
          if (o == null)  // si l'objet o est null
               return false;   // on est sûr qu'ils ne sont pas égaux
          // si les classes des deux objets ne sont pas les mêmes
          if (this.getClass() != o.getClass())[2]
               return false;   // ils ne peuvent être égaux
          A a = (A)o;     // on peut mainternant convertir l'objet en
                               une instance de la classe courante
          return (this.champ1 == a.champ1)    // on compare les champs
               && (this.champ2.equals(a.champ2)     // un à un
               && …;
     }
}
```

Le paquetage `java.util`

On y trouve aussi des classes permettant la représentation de collections d'objets.

La classe `java.util.ArrayList`[3] permet de représenter des collections d'objets sous la forme d'une liste. Son principal avantage par rapport à un tableau est qu'elle se redimensionne automatiquement : en appelant la méthode `add(…)`, la liste est automatiquement redimensionnée pour contenir un nouvel élément. Il existe dans l'API Java d'autres classes permettant de représenter des collections d'objets telles que `java.util.LinkedList` ou `java.util.HashSet`. Elles servent toutes à contenir des éléments. Cependant, les algorithmes d'ajout, d'accès ou de suppression varient d'une structure de données à l'autre. Dans le cadre de ce cours, nous nous limiterons à `ArrayList`. À noter que `java.util.Vector` était auparavant utilisée à la place d'`ArrayList` mais cette classe est depuis considérée comme obsolète.

`ArrayList` est différente des tableaux. En premier lieu, c'est une classe, et il est donc nécessaire de passer par des méthodes pour accéder aux éléments du vecteur (on ne peut pas utiliser les `[]`, il faut utiliser la méthode `get(int)`). De plus, alors qu'il est possible de déclarer des tableaux de n'importe quel type (`int[]`, `Vehicule[]`, etc.), une **`ArrayList`** ne peut contenir que des références à des **`Object`**. Cependant, cette classe étant la racine de l'arbre d'héritage, il est possible de stocker dans un **`ArrayList`** des instances de n'importe quelle classe. La méthode `get(int)` retourne un **`Object`** et il est donc nécessaire d'effectuer une conversion explicite pour pouvoir stocker le résultat de `get(int)` dans la bonne classe. On appelle parfois cette opération un *cast*.

[2] `getClass()` est une méthode héritée de **`Object`** qui renvoie un objet qui représente la classe de l'objet.
[3] https://docs.oracle.com/javase/7/docs/api/java/util/ArrayList.html

```
ArrayList agences = new ArrayList();
agences.add(new Agence(…)); // correct
agences.add(new Bureau(…)); // correct
agences.add(new Object()); // correct /!\
Agence a = agences.get(0); // incorrect
Bureau b = agences.get(0); // incorrect
Object o = agences.get(0); // correct
Agence a = (Agence)(agences.get(0)); // correct, conversion
                                                Object → Agence
Bureau b = (Bureau)(agences.get(0)); // correct, conversion
                                                Object → Bureau
```

On voit clairement à partir de la ligne 5, que la lecture du contenu de la liste risque de poser problème : on ne connaît pas le type des instances stockées dans la liste. Depuis Java 1.5, il est possible de spécifier le type des objets contenus dans la liste au moyens des chevrons (<>). Dans la liste de l'exemple précédent, nous souhaitons stocker que des agences. On imposera à la liste de ne contenir que ce type d'objet lors de la déclaration. Cette méthode a de nombreux avantages parmi lesquels :

- les éléments de la liste seront nécessairement des agences;
- par conséquent, ajouter autre chose qu'un bureau dans la liste crée maintenant une erreur ;
- il est toujours possible d'ajouter des instances de sous-classes de `Agence` ;
- il n'est en revanche pas possible d'ajouter des instances de super-classes de `Agence` ;
- la méthode `get(int)` renvoie une instance de `Agence` au lieu de **`Object`** précédemment ;
- la conversion **`Object`** → `Agence` devient donc inutile.

De nombreuses erreurs uniquement repérables à l'exécution peuvent ainsi être corrigées dès la compilation.

L'exemple précédent devient alors :

```
ArrayList<Agence> agences = new ArrayList<Agence>();
agences.add(new Agence(…)); // correct
agences.add(new Bureau(…)); // correct
agences.add(new Object()); // incorrect
Agence a = agences.get(0); // correct
Bureau b = agences.get(0); // incorrect
Object o = agences.get(0); // correct, conversion Agence → Object
Agence a = (Agence)(agences.get(0)); // correct mais inutile
Bureau b = (Bureau)(agences.get(0)); // correct, conversion
                                                Agence → Bureau
```

Cette déclaration est à préférer par rapport à la déclaration « brute » : correctement utilisée, elle n'a que des avantages. En effet, la déclaration brute n'est aujourd'hui acceptée que pour des raisons de rétro-compatibilité avec les anciennes version de Java : il n'est pas impossible qu'elle soit interdite dans les versions futures. Actuellement, elle indique simplement un avertissement (*warning*). La déclaration « brute » est équivalente à la déclaration d'un **`ArrayList<Object>`**.

Recherche des bytecodes des classes

Soit le fichier `Exemple.java` suivant :

```
import java.util.*;
import projet.location.*;

class A {
     // utilise une classe C
     C inst1 = new C();
     …
}
```

Si on compile ce fichier source depuis le répertoire courant, le compilateur doit trouver le fichier `C.class`. Pour cela, il va considérer successivement que la classe `C` peut appartenir :

- au paquetage par défaut et donc que `C.class` est dans le répertoire courant ;
- au paquetage `java.lang` (recherche de `java/lang/C.class` dans `tools.jar` du JDK). En effet la recherche se fait systématiquement dans le paquetage `java.lang` qui n'a donc pas besoin d'être importé ;
- au paquetage `java.util` ;
- au paquetage `projet.location` : on cherche donc un fichier `projet/location/C.class` dans le répertoire courant.

Option `-classpath` et variable d'environnement `CLASSPATH`

Dans la commande de compilation `javac`, l'option `-classpath` chemin permet d'indiquer un ou plusieurs chemins de recherche (plusieurs chemins sont séparés par '`:`' sous Unix et par '`;`' sous Dos).

Si on lance depuis le répertoire courant :

```
javac -classpath ../ Exemple.java
```

le compilateur va considérer que la classe *C* peut appartenir :

- au paquetage par défaut et donc la chercher dans le répertoire père du répertoire courant (chemin `../`). Attention on ne cherche plus dans le répertoire courant ;
- au paquetage `java.lang` ;
- au paquetage `java.util` ;
- au paquetage projet.location : on cherche donc un fichier `projet/location/C.class` dans le répertoire père du répertoire courant (chemin `../`).

On indiquerait `javac -classpath ../:./ Exemple.java` pour chercher dans le répertoire courant et dans son père.

De la même manière, `java -classpath` chemin test cherche, dans le répertoire indiqué par chemin les classes et ressources nécessaires pour exécuter le programme principal de la classe `Test`.

La variable d'environnement `CLASSPATH` permet d'énumérer les chemins d'accès aux classes qui devront être localisées.

Paquetages et niveaux de visibilité

Pour une classe ou une interface :

Modificateur	Visibilité	
aucun	paquetage	Accessible seulement dans son paquetage
`public`	publique	Accessible de partout

Pour un champ (attribut, méthode ou classe interne) d'une classe `A` :

Modificateur	Visibilité	
`private`	privé	Accessible seulement depuis sa propre classe
aucun	paquetage	Accessible seulement dans le paquetage de `A`
`protected`	protégé	De partout dans le paquetage de `A` et si `A` est public dans les classes héritant de `A` dans les autres paquetages
`public`	publique	Accessible de partout dans le paquetage de `A` et partout ailleurs si `A` est public