

TD2 : Rappels JAVA – Aspects avancés

Ce TD a pour objectif de parcourir les différentes notions et notations OO en Java. La plupart ont été vues en 1^{ère} mais nous les reprenons ici afin de vous rafraîchir la mémoire.

Rappels théoriques

Classes et protection des données

Le Java ne nécessite aucun prototype. Toutes les méthodes sont codées dans le corps de la classe. Java reconnaît quatre niveaux de protection de données au travers de trois mots réservés, utilisables pour les attributs, les fonctions membres mais également les classes. Ces mots réservés doivent être placés devant chaque attribut ou méthode.

- **public** : le membre est accessible à tous.
- **protected** : le membre est accessible par la classe et ses classes héritée.
- **private** : le membre est accessible uniquement via la classe.
- **(rien)** : le niveau de protection par défaut du Java est dit « package », c'est à dire qu'il limite l'accès aux autres classes du package.

Une classe peut être public ou « package ». A noter que si le Java ne reconnaît pas de paramètre par défaut, il permet par contre d'appeler un constructeur à partir d'un autre, à l'aide de la référence **this**.

Héritage

L'héritage en java se fait à l'aide du mot réservé **extends** suivi du nom de la classe héritée. Si aucune classe n'est indiquée, la classe hérite automatiquement d'`Object`. L'héritage multiple n'existe pas en Java (voir toutefois la notion d'interface, plus loin), de même que l'héritage privé ou protégé. Au sein d'une classe fille, une méthode de la classe mère peut être appelée à l'aide de l'instruction **super** suivie du nom de la méthode. Le constructeur de la classe mère est appelé de même manière, avec juste **super** suivi des éventuels paramètres. **super** fourni en fait une simple référence sur la classe mère.

Classe Object

La classe `java.lang.Object` est la mère commune de toutes les classes Java. Elle ne contient pas de champs, mais une série de méthodes d'utilisation courante. La plupart sont essentiellement présentes afin de garantir que chaque objet les implémente. Toutefois, il est souvent nécessaire de les surcharger. Les plus utilisées de ces méthodes sont reprises ici, avec leurs signatures, leur

fonction et leur implémentation dans `Object`.

Signature	<code>public String toString()</code>
Fonction	Renvoie la représentation de l'objet sous forme d'une chaîne.
Object	Renvoie le nom de la classe et une forme d'identifiant (hashcode) de l'objet

La méthode `toString()` est implicitement appelé à chaque fois qu'une conversion entre un objet et un `String` doit être réalisée, par exemple en vue de l'envoi dans un flot. Elle devrait être surchargée pour tout objet.

Signature	<code>protected Object clone()</code>
Fonction	Retourne une copie de l'objet
Object	Id.

La méthode `clone()` est rendue indispensable par le mécanisme de référence du Java. En effet, le code suivant ne fait pas de copie des champs, mais fait tout simplement pointer `d1` et `d2` vers le même objet.

```
Date d1 = new Date(1999,10,12);  
Date d2 = d1;
```

Un appel explicite doit donc être fait lorsqu'il faut travailler avec une copie. De manière générale, la méthode `clone()` crée un nouvel objet de même type, et y copie les différentes valeurs des champs de la source.

Signature	<code>public boolean equals(Object o)</code>
Fonction	Retourne true si <code>o</code> réfère un objet identique à <code>this</code>
Object	Id.

Comme le Java ne permet pas de surcharge d'opérateurs, l'opérateur de comparaison « `==` » fait systématiquement une comparaison sur les références, c'est à dire qu'il ne renvoie vrai que si les deux références pointent vers le même objet. Pour comparer deux objets par rapport à leur contenu, il faut utiliser la méthode `equals (Object)` qui peut (et doit), elle, être surchargée.

Collections

Le package `java.util` fournit une série de classes représentant des collections d'objets et fournissant les méthodes attendues sur ce type de structures. Depuis la version 1.5 de Java, la généricité permet d'indiquer le type des éléments..

Pour en savoir plus : <http://java.sun.com/docs/books/tutorial/collections/index.html>

Polymorphisme

Le Java est polymorphe par défaut. Si on appelle une méthode sur un objet, Java tient compte du type réel de l'objet et pas de son type déclaré.

Classes abstraites

Une classe qui ne peut être instanciée est déclarée au moyen du mot réservé **abstract** devant son nom. Toute classe qui déclare une ou plusieurs méthodes sans les définir (prototype seul) devra obligatoirement être déclarée abstraite. Toutes les classes filles devront impérativement implémenter toutes les méthodes déclarées, ou être elles-mêmes déclarées abstraites.

Interfaces

Les interfaces (à ne pas confondre avec les interfaces *graphiques*) sont une structure syntaxique propre au Java et capitales dans ce langage. Il s'agit techniquement de « classe abstraites pures », c'est à dire de classe ne contenant que prototypes et éventuellement des constantes statiques, sans aucune implémentation.

Conceptuellement, une interface fournit un « contrat », c'est à dire une série de fonctionnalités qu'une classe s'engage à fournir, sous forme d'une série de signatures de méthodes. L'intérêt pour l'utilisateur d'une classe implémentant (mot réservé **implements**) une interface précise est d'être assuré d'y retrouver toutes les méthodes déclarées dans l'interface. Une interface fournit en quelque sorte les services publics des classes qui l'implémentent. Une classe ne peut hériter que d'une seule autre, mais peut implémenter un nombre quelconque d'interfaces. Une interface peut elle même hériter (**extends**) d'une ou plusieurs autres interfaces. Par exemple, l'interface `java.util.Collection` définit tous les services qui doivent être prévus par une collection, sous forme d'une série de signatures permettant l'ajout d'objets, l'accès à un itérateur ou à la taille actuelle de la collection. `ArrayList` et `Vector` sont deux collections implémentant `java.util.Collection`.

Applications

Pour tous ces exercices, nous vous demandons d'écrire une Javadoc complète.

1. MonCompte

Créez une classe `MonCompte` représentant un compte bancaire.

Les propriétés en seront `int agence`, `int compte`, `int contrôle`, `String libellé`.

Le constructeur sera de la forme `MonCompte(int agence, int compte, int contrôle, String libellé)` et pourra envoyer une erreur de type `MonCompteException` lorsque `agence`, `compte`, `contrôle` reprennent un nombre inférieur à 0 ou respectivement supérieur à 999, à 9999999, à 99. Une erreur analogue sera générée si le libellé est null.

Les méthodes à surcharger seront `toString()` [compte sous la forme 000-0000000-00 suivi du libellé], `equals(Object o)` qui doit renvoyer `true` si les trois entiers formant le n° des deux comptes sont respectivement égaux, et `clone()`.

Les méthodes à fournir seront `String getNumCpt()` renvoyant le n° de compte sous la forme 000-0000000-00, `int getCrc()` qui renvoie le reste de la division entière du nombre représenté par les 10 premiers chiffres du n° de compte divisé par 97 (si ce nombre vaut 0, il faudra renvoyer 97), `boolean isValid()` renvoyant `true` si `getCrc()==contrôle`, `false` sinon.

Mettez en oeuvre Junit pour tester la validité de `getCrc()` et `isValid()`.

2. ListeCompte

Créez une classe `ListCompte`. Cette classe container servira à mémoriser des comptes. Utilisez un `Vector` comme container interne. Votre classe va-t-elle hériter d'une collection ou composer une collection ? Écrivez une méthode de test qui itère sur la liste des comptes afin de vous remémorer ce concept. (pour l'itération, en première phase utilisez un `Iterator` comme vu précédemment puis utilisez un `for each`).

3. CRCable

Créez une interface *CRCable*. Cette interface certifie que la classe l'implémentant peut fournir un code CRC pour chacun de ses objets, permettant par exemple de vérifier si un changement a eu lieu. L'interface déclare deux méthodes: `int getCrc()` et `boolean isValide()`. Reprenez votre classe *MonCompte* et implémentez cette interface. Créez ensuite de toute pièce une classe *CommunicationStructurée* (3 zones de respectivement 3, 4 et 5 chiffres, les deux derniers jouant le rôle de crc (algorithme identique au précédent) pour les 10 premiers) implémentant également l'interface. Enfin, créez une méthode statique recevant un vecteur de *CRCable* et affichant pour chacun l'objet, son code CRC et sa validité.

4. RandomCollection

Créez une interface *RandomCollection<Object>* héritant de *Collection*, mais déclarant une nouvelle méthode `Object getRandom()` renvoyant un élément de la collection pris au hasard. Créez une classe *RandomArrayList* qui implémente cette interface (prenez la classe *ArrayList* comme base).