

Types Abstraits & Base de la POO - TD 3



Conception Objet UML & Programmation Objet Java Le carnet d'adresses

I - Cahier des charges & conception

On souhaite ici réaliser une petite application “Carnet d'adresses” permettant de gérer un ensemble de contacts. Un contact pourra être de deux types : personne ou entreprise. Dans tous les cas, un contact possédera :

- un nom
- un numéro de téléphone portable
- un numéro de téléphone fixe
- une adresse postale pouvant être en France ou à l'étranger

Dans le cas d'une entreprise, on ajoutera son numéro de siret et sa catégorie parmi artisan, commerce ou concessionnaire.

Dans le cas d'une personne, on ajoutera :

- un prénom
- une date de naissance
- un age

Concernant les adresses, on distinguera deux catégories :

- les adresses européennes définies par :
 - un numéro de rue
 - un nom de rue

- un code postal
- et un nom de ville

- les adresses US définies par :

- un numéro de rue,
- un nom de rue,
- un état,
- un code zip,
- un nom de ville

Un carnet de contacts sera donc composé :

- d'un nom
- d'un ensemble de contacts
- du nombre de contacts enregistrés

Il devra permettre :

- l'ajout d'un contact
- de lister l'ensemble des contacts à l'écran,
- de rechercher un contact par son nom
- de supprimer un contact

Exercice I.1

Prise en main d'UmlDesigner

Dans un premier temps, mettons en place le logiciel UMLDesigner.

1. Récupération de l'archive UML Designer, puis décompression sur le disque.
2. Au même niveau d'arborescence que le répertoire Workspace-TAP, créer un dossier Workspace-UML.
3. Lancement d'UML Designer
4. Définition de votre Workspace ; indiquer l'emplacement de celui créé précédemment
5. Création d'un premier projet pour réaliser notre diagramme de classe UML.
 - New UML Project : Carnet-Contacts
 - Créer un nouveau modèle CarnetContact
 - Clic droit sur <Model> NewModel sous carnet-contacts.uml puis “New Representation” puis “NewModel Class Diagram” : carnet-contacts-diag-classes.

Exercice I.2

Carnet de contacts : diagramme de classes

1. Dans le projet précédemment créé, réaliser la conception du diagramme de classes du projet carnet de contacts. On organisera les différentes classes dans 3 espaces de nommage différents :
 - carnet-contacts
 - contacts
 - adresses

II - De la conception à l'implémentation

Nous étant maintenant mis d'accord sur la conception de notre petite application, nous pouvons maintenant nous concentrer sur la phase d'implémentation, consistant en la traduction de cette dernière dans le langage Java.

Avant toute chose, nous allons prendre en main l'environnement de développement Java.

Exercice II.1

Prise en main de l'environnement Java

Dans un premier temps nous utiliserons l'éditeur *Visual Studio Code*.

Après avoir créé un nouveau workspace sur votre disque, intégrer le dans l'éditeur.

1. Créer un répertoire `carnetcontacts` qui servira de base à notre petit projet. Plus précisément, ce répertoire sera le *package* racine de celui-ci.
2. Afin de séparer le résultat de la compilation de nos fichiers sources, créer un répertoire `target` à l'intérieur du répertoire précédent. C'est dans celui-ci que nous indiquerons au compilateur de créer les fichiers `.class`.
3. Dans le répertoire `carnetcontacts`, ajouter un fichier `App.java` qui contiendra :

```

1 package carnetcontacts;
2
3 public class App {
4     public static void main(String args[]){
5         System.out.println("Salut les cocos !!!");
6     }
7 }
8 
```

4. Dans un terminal, et à la racine de votre répertoire workspace, compiler le fichier par la commande :
`javac -d carnetcontacts/target carnetcontacts/App.java`
5. Lancer l'exécution de la classe `App` en tapant : `java carnetcontacts.App`
Vous devriez obtenir l'erreur suivante :

```

Error: Could not find or load main class carnetcontacts.App
Caused by: java.lang.ClassNotFoundException: carnetcontacts.App

```

Ceci est tout à fait normal. En effet, il nous faut indiquer à l'environnement java le(s) chemin(s) dans le(s)quel(s) il trouvera notre(nos) classe(s). Ceci porte le nom de *classpath*.

6. Relancer l'exécution en utilisant cette fois la commande suivante :
`java -cp ./carnetcontacts/target:. carnetcontacts.App`
7. Enrichissons maintenant notre projet d'un nouveau package et d'une nouvelle classe :
 - (a) dans le dossier `carnetcontacts` créer un dossier `monpack` correspondant à un nouveau package.
 - (b) dans ce dossier ajouter le fichier `MaClasse.java` qui contiendra le code suivant :

```

1 package carnetcontacts.monpack;
2
3 public class MaClasse {
4     private String laChaine;
5
6     public MaClasse(final String ch){
7         laChaine = ch;
8     }
9
10    public String getLaChaine() {
11        return laChaine;
12    }
13 }
14 
```

- (c) Modifier le fichier `App.java` avec le code suivant :

```

1 package carnetcontacts;
2 import carnetcontacts.monpack.MaClasse;
3
4 public class App {
5     public static void main(String args[]) {
6
7         System.out.println("Salut les cocos !!!");
8         MaClasse mc = new MaClasse("Salut les cocos depuis la classe !!!");
9         System.out.println(mc.getLaChaine());
10    }
11 }
12

```

8. La compilation se déroulera comme précédemment à la différence qu'il faudra indiquer au compilateur les différentes fichiers sources à traiter :

```
javac -d carnetcontacts/target carnetcontacts/App.java carnetcontacts/monpack/MaClasse.java
```

La commande suivante produit le même résultat, mais évite de lister l'ensemble des fichiers sources :

```
javac -d carnetcontacts/target -sourcepath carnetcontacts **/*.java
```

9. Tester cette nouvelle version en utilisant la commande d'exécution précédente.

Exercice II.2

Carnet de contacts : implémentation Java

Vous voilà maintenant armé pour réaliser l'implémentation de la conception réalisée précédemment dans ce TD.

1. Implémenter l'ensemble de votre conception, en respectant au mieux les choix (association, cardinalité, ...) faits dans celle-ci.
2. Dans la classe App.java, tester votre implémentation en utilisant les différentes fonctionnalités proposées.
3. Dans toutes vos classes, ajouter, si nécessaire, les redéfinitions des méthodes toString(), equals() et clone()
4. Tester les nouveaux comportements obtenus.

Comme nous l'avons vu durant le cours, la redéfinition de la méthode equals() d'une classe nécessite, afin de conserver un comportement homogène de la classe¹, la redéfinition de la méthode hashCode(). Nous présentons ci-dessous le principe général de l'algorithme de redéfinition de cette méthode.



Les méthodes equal & hashCode :

Nous avons vu précédemment la nécessité, dans de nombreux cas, de redéfinir la méthode equal. Il est alors nécessaire d'également redéfinir la méthode hashCode car celle-ci est implicitement liée à la première.

En effet, la propriété suivante doit toujours être vérifiée pour maintenir un fonctionnement homogène des instances de la classe :

Si obj1.equals(obj2) est vrai, alors obj1.hashCode == obj2.hashCode doit être vrai également.

Méthode d'écriture de la fonction hashCode :

- après initialisation du hashCode^a, le calcul de celui-ci doit prendre en compte la valeur de chacun des attributs mis-en-jeux dans la méthode equals. Soit un attribut att :
 - att de type boolean : renvoyer 0 ou 1
 - att de type entier (byte, char, short ou int) : prendre directement la valeur
 - att de type float : conversion en int (méthode Float.floatToIntBits(att)).
 - att de type complexe (i.e.classe) : utilisation de la méthode att.hashCode() si att ≠ null, 0 sinon.
 - att de type collection : traitement de tous les éléments de la collection en appliquant les règles précédentes.
- chacun des nouveaux hashCodes obtenus est ajouté au précédent après avoir multiplié ce dernier par un nombre constant^b.

a. Avec un nombre premier de préférence pour éviter les problèmes de multiple

b. dont la valeur peut aussi être un nombre premier, mais différent de celui choisi pour l'initialisation de la valeur du haschode de départ

1. En particulier lors de l'utilisation dans une collection

5. Dans toutes les classes dans lesquelles vous avez redéfini la méthode `equals()` ajouter une redéfinition de la méthode `hashCode` en respectant le principe ci-dessus.

Pour finir, nous allons mettre en œuvre la documentation d'API de notre petit projet.

6. Dans chacune des classes du projet carnet-contact, ajouter la documentation technique pour tous les membres publics. Vous utiliserez les annotations vues en cours.
7. La génération de la documentation se fait à l'aide de l'utilitaire `javadoc` du JDK.
 - (a) avant toute chose, créer le répertoire `doc` à la racine de votre répertoire projet (normalement `carnetcontacts`)
 - (b) dans le terminal, à la racine de votre workspace, exécuter la commande `javadoc` suivante :

```
javadoc -d carnetcontacts/doc -subpackages carnetcontacts
```
 - (c) ouvrir le fichier `index.html` produit et analyser la documentation générée

‡ ‡ ‡