# Types Abstraits & Base de la POO - TD 4

‡ ‡ ‡

# Conception Objet UML & Bases de la Programmation Objet Java
# La Deque en Java...

## I - Cahier des charges & conception

On souhaite ici réaliser une bibliothèque permettant la manipulation d'une deque générique.

*Exercice I.1*

### La deque : diagramme de classes

1. À l'aide du plugin plantUml de VisualStudioCode, réaliser la conception du diagramme de classe du projet Deque.

## II - De la conception à l'implémentation

Nous étant maintenant mis d'accord sur la conception de notre deque, 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

Nous utiliserons, là encore, l'éditeur *Visual Studio Code*.
Après avoir créé un nouveau répertoire `Td04-Deque-Java` sur votre disque, intégrez-le dans votre workspace.

1. Créer un répertoire `dequejava` 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 `dequejava`, ajouter un fichier `App.java` qui contiendra :

```java
package dequejava;

public class App {
  public static void main(String args[]){
    System.out.println("Salut les cocos !!!");
  }
}
```

Listing 1 – Fichier App.java

4. Dans un terminal, et à la racine de votre répertoire `Td04-Deque-Java`, compiler le fichier par la commande :
`javac -d dequejava/target dequejava/App.java`
5. Lancer l'exécution de la classe App en tapant : `java dequejava.App`
Vous devriez obtenir l'erreur suivante :

```
Error: Could not find or load main class dequejava.App
Caused by: java.lang.ClassNotFoundException: dequejava.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 ./dequejava/target:. dequejava.App`
7. Enrichissons maintenant notre projet d'un nouveau package et d'une nouvelle classe :
   (a) dans le dossier `dequejava` créer un dossier `monpack` correspondant à un nouveau package.
   (b) dans ce dossier ajouter le fichier `MaClasse.java` qui contiendra le code suivant :

```java
package dequejava.monpack;

public class MaClasse {
  private String laChaine;
```

```java

  public MaClasse(final String ch){
    laChaine = ch;
  }

  public String getLaChaine() {
    return laChaine;
  }
}
```

Listing 2 – Fichier MaClasse.java

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

```java
package dequejava;
import dequejava.monpack.MaClasse;

public class App {
  public static void main(String args[]){

    System.out.println("Salut les cocos !!!");
    MaClasse mc = new MaClasse("Salut les cocos depuis la classe !!!");
    System.out.println(mc.getLaChaine());
  }
}
```

Listing 3 – Fichier App.java - V2

8. La compilation se déroulera comme précédemment à la différence qu'il faudra indiquer au compilateur les différents fichiers sources à traiter :
   `javac -d dequejava/target dequejava/App.java dequejava/monpack/MaClasse.java`
   La commande suivante produit le même résultat, mais évite de lister l'ensemble des fichiers sources :
   `javac -d dequejava/target -sourcepath dequejava **/*.java`
9. Tester cette nouvelle version en utilisant la commande d'exécution précédente.

*Exercice II.2*

## La deque : 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 classe [1], 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)).

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

— att de type complexe (i.e.classe) : utilisation de la méthode att.hashCode() si att $\neq$ 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

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

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

6. Dans chacune des classes du projet dequejava, ajouter la documentation technique pour tous les membres publiques. 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 `dequejava`)
   (b) dans le terminal, à la racine de votre workspace, exécuter la commande `javadoc` suivante :
   `javadoc -d dequejava/doc -subpackages dequejava`
   (c) ouvrir le fichier `index.html` produit et analyser la documentation générée

‡ ‡ ‡