Exercice 1 (API Java) :

Écrire la classe `Ensemble` comme étant une collection d'éléments ne contenant pas de doublon. Elle sera donc implémentée à l'aide d'un `Vector` d'`Object` et de ses méthodes.
Cette classe doit permettre l'utilisation des opérations suivantes :

1. `estVide()`, qui retourne `true` si l'ensemble est vide, `false` sinon.
2. `taille()` , qui retourne le nombre d'éléments dans l'ensemble.
3. `contient()`, qui retourne `true` si l'élément passé en paramètre appartient à l'ensemble, `false` sinon.
4. `obtenir()`, qui renvoie l'élément situé à l'emplacement envoyé en paramètre.
5. `ajouter()`, qui ajoute un élément à l'ensemble.
6. `copie()`, qui retourne un ensemble contenant les mêmes éléments.
7. `retournerEnlever()`, qui retourne un élément de l'ensemble et qui l'enlève de l'ensemble (si l'ensemble n'est pas vide).
8. `intersection()`, qui renvoie un nouvel ensemble ne contenant que les éléments contenus à la fois dans l'ensemble courant et dans l'ensemble envoyé en paramètre.
9. Écrire un programme pour tester votre implémentation.
10. Modifier la classe `Bureau` en utilisant la classe `Ensemble` dans le package `outils` plutôt que la classe `Vector` pour représenter l'ensemble des garages liés à un bureau.
11. Ajouter une méthode `contient()` à `Bureau` qui renvoie `true` si le bureau contient le garage envoyé en paramètre et `false` sinon.
12. Dans la fonction `main()`, créer deux garages identiques `g0` et `g0b` (tous leurs attributs sont équivalents). Ajouter `g0` à un bureau et vérifier si ce bureau contient `g0b`.
13. Redéfinir la méthode `equals(Object o)` pour la classe `Garage` de telle manière que `g0` et `g0b` soient équivalents et revérifier si le bureau contient `g0b`.

Exercice 2 (Paquetages) :

1. Écrire la classe représentant l'`Entreprise` de location comme elle est proposée dans l'exercice 2 du TD3 en utilisant la classe `Ensemble`.
2. Effectuer les modifications demandées dans les exercices 2 et 3 du TD 3 (champs privés, classe abstraite, variable de classe…).
3. Modifier l'organisation des fichiers écrits jusque là et concernant l'application "Location de Véhicules". On créera un répertoire *TPVehicule* qui contiendra un répertoire *mobiles* contenant les classes `Vehicule`, `Voiture`, `Camion` et `Autocar`, un répertoire *business* contenant les classes `Entreprise`, `Agence`, `Bureau` et `Garage` et un *répertoire* outils contenant la classe `Ensemble`.
4. Réaliser différents fichiers de tests (méthode `main`) placés soit dans `TPVehicule`, soit dans un des sous répertoires précédents et voyez comment vous devez demander la compilation et l'exécution.

Exercice 3 (Javadoc) :

1. Commenter au moins deux classes du TP 1 (comme dans l'exemple donné ci-dessous), générer la documentation correspondante et consulter le résultat obtenu.

Annexes

I. 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.*;`

Recherche des bytecode 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 publique dans les classes héritant de A dans les autres paquetages
public	publique	Accessible de partout dans le paquetage de A et si A publique de partout ailleurs

II. Javadoc

Lorsqu'on développe un paquetage ou un ensemble de classes, l'utilitaire du JDK (Java Development Toolkit) **Javadoc** permet de générer automatiquement une documentation des classes au format html à partir d'une analyse du code source et des commentaires inclus dans les fichiers source.
Pour cela les commentaires doivent impérativement commencer par **/**** et se terminer par ***/**.
Rappelons que // peut aussi servir pour des commentaires sur une seule ligne et que /* et */ encadrent un commentaire qui peut s'étendre sur plusieurs lignes,

```
/**
    Ce commentaire sert à décrire la classe MaClasse.
    Il doit être placé juste avant la déclaration.
*/
public class MaClasse{
  /** Commentaire pour l'attribut monAttribut */
  private int monAttribut;
  ...
  /** Commentaire pour la méthode maMethode
   */
  public void maMethode(){
   ...
}
```

Des instructions commençant par le symbole **@** permettent d'enrichir les informations traitées par javadoc :

```
/** Dans l'entête de la classe :
 *  @author Toto
 *  @version 1.0.2
 */

//on trouve ici les commentaires de la méthode maMethode définie
//juste en-dessous
/** Description des paramètres d'une méthode :
    @param x pour décrire le paramètre x, e
    @param y pour le paramètre y.*/
  public void maMethode(int x, String y, Vehicule z){
  ...
```

Citons également **@deprecated** pour indiquer qu'une méthode est dépréciée, et **@exception** qui indique que la méthode peut générer une exception.

L'appel à javadoc est fait de la façon suivante :

javadoc [-author] [-version] [-d ./doc] *.java

ou :

javadoc [-author] [-version] [-d ./doc] package1 package2 …

`-d ./doc` permet de spécifier le répertoire[1] où sera mise la documentation.
La documentation est constituée d'un ensemble de fichiers html et elle est accessible à partir du fichier **index.html**. Remarque: les tags html comme   sont utilisables pour insister sur certaines parties de textes.

[1] Le répertoire ./doc doit exister.

```java
import java.util.Date;

/**
 * La classe Vehicule décrit les propriétés communes aux véhicules que
 * l'on veut manipuler,
 * @author B. Duval
 * @version TP numero 3
 */
public class Vehicule {
      /**
       * Tous les attributs sont privés donc ils n'apparaîtront pas
       * dans la doc car ils ne sont pas visibles à
       * l'extérieur de la classe,
       */
      private String modele;
      private Date dateAchat;
      private float prixAchat;
      private String numeroImmatriculation;
      private String permis;

      public Vehicule(String modele, Date dateAchat, float prixAchat,
String numeroImmatriculation, String permis) {
       this.modele = modele;
       this.dateAchat = dateAchat;
       this.prixAchat = prixAchat;
       this.numeroImmatriculation = numeroImmatriculation;
       this.permis = permis;
      }

      /**
       * La méthode afficher s'appuie sur la méthode toString que l'on
       * redéfinit pour chaque classe afin d'avoir les informations
       * souhaitées pour chaque type de véhicule,
       * @see #toString()
       */
      public void afficher() {
       System.out.println(this);
      }

      /**
       * @param vol un volume donné
       * @return vrai si ce véhicule peut transporter un volume égal à vol
       */
      public boolean peutTransporterVolume(int vol) {
       return false;
      }

      public String toString() {
       return "Modèle : " + modele + "\nImmat. : " +
numeroImmatriculation + "\nCout : " + coutLocation();
      }
}
```

III. API Java

La bibliothèque standard de classes JAVA est accessible en ligne sur :
http://docs.oracle.com/javase/6/docs/api/
Elle est organisée en paquetages thématiques :
- java.lang : classes de base du langage (chaines, math, 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, évènements, …)
- 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*. Toute classe hérite de *Object*. (voir la liste des méthodes de cette classe).
On a déjà vu la méthode *toString()* qu'il est intéressant de redéfinir pour retourner une chaîne de caractères adaptée à la classe de l'objet traité.

Egalité 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 objets 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 2 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 :

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.

La classe java.lang.Math

*public final class **Math***
extends Object
The class Math contains methods for performing basic numeric operations such as the elementary exponential, logarithm, square root, and trigonometric functions

contient deux constantes

*public static final double **E***
*public static final double **PI***

et des méthodes statiques.

Le paquetage java.lang contient les classes enveloppes(wrappers) associées aux types simples: classe *Double* associée à *double*, class *Integer* associée à *int*…

Le paquetage `java.util`

Nous avons déjà utilisé la classe *Date* de ce paquetage. On y trouve aussi des classes permettant la représentation de collections d'objets.

La classe `java.util.Vector`[2] permet de représenter des collections d'objets sous la forme d'un vecteur (tableau)[3]. Cette classe est cependant 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()`). De plus, alors qu'il est possible de déclarer des tableaux de n'importe quel type (`int[]`, `Vehicule[]`, etc.), un vecteur 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 `Vector` des instances de n'importe quelle classe.

```
java.util.Vector v = new java.util.Vector();
v.add(new Bureau(…));
```

La méthode `get()` retourne un `Object`, et il est donc nécessaire d'effectuer une conversion de type explicite pour pouvoir stocker le résultat de `get()` dans une référence vers une sous-classe d'`Object`, ou appeler une méthode d'une sous-classe d'`Object`.

```
Object o = v.get(0); // correct
Bureau b = v.get(0); // faux, un Bureau ne peut recevoir un Object
Bureau b = (Bureau)(v.get(0)); // correct
```

Mais le principal avantage d'un vecteur par rapport à un tableau est qu'il se redimensionne automatiquement : en appelant la méthode `add()`, le vecteur est automatiquement redimensionné pour contenir un nouvel élément.

[2] Consulter l'aide en ligne de l'API Java pour une description complète des méthodes de cette classe.
[3] Il existe dans l'API Java d'autres classes permettant de représenter des collections d'objets telles que `java.util.ArrayList` ou `java.util.HashSet`.