TD JAVA n°4: Collection et Généricité

Exercice 1 : Manipulation des Collections : Comparaison et tri des objets …

Considérons la classe `Montagne` définie de la manière suivante :

```
public class Montagne {
      private String nom;
      private int hauteur;

      //Constructeurs
      public Montagne(){}

      public Montagne(String n, int h)
      { this.setNom(n); this.setHauteur(h);}

      // Getteurs & Setteurs
      public String getNom()
      {return this.nom;}

      public void setNom(String nouvNom)
      {this.nom=nouvNom;}

      public int getHauteur()
      {return this.hauteur;}

      public void setHauteur(int nouvHaut)
      {this.hauteur=nouvHaut;}

      //toString
      public String toString(){
      return nom + " \t --> "+ hauteur + " m d'altitude";}
}
```

1. Dans un premier temps, écrire un programme **`TestTriMontagnes.java`** correspondant au jeu d'essai suivant :
   - Instanciation d'une collection `lesMontagnes` (de type `ArraList` par exemple) contenant des objets de type `Montagne`.
   - Ajout dans cette collection d'objets de type `Montagne`
   - Affichage du contenu de la collection `lesMontagnes` créée.

```
--- Des Montagnes ---
Mont Blanc        --> 4807 m d'altitude
Puy de Sancy      --> 1886 m d'altitude
Puy de Dôme       --> 1464 m d'altitude
Pic du Midi       --> 2877 m d'altitude
Pic d'Aneto       --> 3404 m d'altitude
Pic du Canigou    --> 2784 m d'altitude
```

2. Que pensez-vous des déclarations suivantes ?

`ArrayList<Montagne> lesMontagnes = new ArrayList<Montagne>();`	☐ Correct ☐ Erreur de compilation
`List<Montagne> lesMontagnes = new ArrayList<Montagne>();.`	☐ Correct ☐ Erreur de compilation
`Collection<Montagne> lesMontagnes = new ArrayList<Montagne>();`	☐ Correct ☐ Erreur de compilation

Et de celles-là ?

`List<Montagne> lesMontagnes = new List<Montagne>();.`	☐ Correct ☐ Erreur de compilation
`ArrayList<Montagne> lesMontagnes = new List<Montagne>();`	☐ Correct ☐ Erreur de compilation

3. Tri de la collection suivant un seul critère.

3.1 Retrouvez dans le cours, l'instruction permettant d'effectuer un tri sur les collections (transparent n°29), et adaptez cet appel à notre collection `lesMontagnes`.

3.2 ***Tri des montagnes par nom :*** à partir de la méthode `sort` proposée dans le cours (et extraite de l'API) :

`public static <T extends Comparable<? super T>> void sort(List<T> list)`

Sorts the specified list into ascending order, according to the *natural ordering* of its elements. All elements in the list must implement the `Comparable` interface. Furthermore, all elements in the list must be *mutually comparable* (that is, `e1.compareTo(e2)` must not throw a `ClassCastException` for any elements `e1` and `e2` in the list).

Modifier la classe `Montagne` pour pouvoir utiliser l'instruction de la question 3.1 et effectuer à l'aide de cette instruction un ***tri des montagnes par nom***

4. Tri de la collection suivant plusieurs critères : *Tri des montagnes par nom et par hauteur :*

Nous venons de rendre `Comparable` un objet `Montagne` ce qui nous permet de trier notre collection…mais limite notre tri au nom des montagnes puisqu'on n'a qu'une seule possibilité d'implémenter la méthode `compareTo`…

Nous allons voir dans cette question comment trier la collection suivant plusieurs critères (le `nom` et la `hauteur`) en utilisant une autre méthode statique **`sort`** à 2 paramètres proposée également dans l'API de la classe `Collections`.

java.util

Class Collections

java.lang.Object
  └ **java.util.Collections**

`public static <T> void sort(List<T> list, Comparator<? super T> c)`

Sorts the specified list according to the order induced by the specified comparator. All elements in the list must be *mutually comparable* using the specified comparator (that is, `c.compare(e1, e2)` must not throw a `ClassCastException` for any elements `e1` and `e2` in the list).

Cette **nouvelle méthode `sort` *surchagée*** accepte en tant que second paramètre un objet de type **`Comparator`**.

java.util

`public interface Comparator<T>`

Interface Comparator<T>

A comparison function, which imposes a *total ordering* on some collection of objects. Comparators can be passed to a sort method (such as `Collections.sort`) to allow precise control over the sort order. Comparators can also be used to control the order of certain data structures (such as `TreeSet` or `TreeMap`).

Method Summary

int	**compare**(T o1, T o2) Compares its two arguments for order.
boolean	**equals**(Object obj) Indicates whether some other object is "equal to" this Comparator.

Remarque : Lors de l'implémentation de l'interface `Comparator`, il n'est pas nécessaire de définir la méthode **`equals`**.
En effet la méthode `equals` est déjà définie dans la classe `Object`. Il est utile de définir **`equals`** (donc redéfinir la méthode de la classe `Object`) que si vous jugez que votre méthode (`equals`) est plus efficace et donc rapide.

Contentons-nous donc de la méthode `compare` !!!
Comme la méthode `compareTo`, la méthode **`compare`** retourne une valeur entière indiquant l'ordre des objets soumis :

- ➔ Un entier négatif signifie que le premier argument est inférieur au second.
- ➔ Une valeur égale à zéro signifie que le premier argument est égal au second.
- ➔ Un entier positif signifie que le premier argument est supérieur au second.

compare

`int compare(T o1, T o2)`

Compares its two arguments for order. Returns a negative integer, zero, or a positive integer as the first argument is less than, equal to, or greater than the second.

*Pour trier la collection `lesMontagnes` **par nom** en appelant la méthode `sort` surchargée, il est nécessaire de définir un nouveau comparateur…*

4.1 Ecrire une classe **`ComparateurNom`** qui implémente l'interface **`Comparator`** sur des objets de type **`Montagne`** et permet de trier ces objets (`Montagne`) suivant leur **`nom`**.

Bon à savoir pour le TP …
Il est d'usage d'implémenter les comparateurs "à la volée" dans les classes qui les utilisent.
*Nous implémenterons donc **`ComparateurNom`** dans le fichier **`TestTriMontagnes`** après la déclaration de `TestTriMontagne`…*

```
public class TestTriMontagnes {  …  }
```

Comme il n'y peut y avoir qu'une seule classe publique par fichier, nous écrirons :

```
class ComparateurNom ..................... {…}
```

4.2 Ecrire les instructions permettant de trier la liste de montagnes par nom (`lesMontagnes`) avec un appel à la méthode `sort` surchargée à 2 paramètres.

4.3 De la même manière, mettre en place une classe **`ComparateurHauteur`** qui permet de trier les montagnes suivant leur altitude, et écrire le tri à l'aide de la méthode sort et d'un objet de cette classe.

En résumé :

- ➔ L'appel de la méthode `sort` à ***1 seul argument*** signifie que c'est la méthode **`compareTo()`** de l'élément qui détermine l'ordre. Les éléments de la liste DOIVENT donc implémenter l'interface **`Comparable`**
- ➔ L'appel de la méthode `sort` à ***2 arguments*** signifie que c'est la méthode **`compare()`** du comparateur (objet de type **`Comparator`**) qui sera utilisée pour déterminer l'ordre des éléments. Si vous choisissez ce cas-là, vous n'avez plus besoin d'implémenter `Comparable.` La classe `Comparator` est externe au type d'élément comparée. C'est une classe séparée, il sera donc possible de créer autant de comparateurs que vous le souhaitez.

Exercice 2 : Cabinet Médical : Introduction à la notion d'architecture logicielle (pattern MVC /DTO /DAO)

Jusqu'à maintenant nous programmions directement les essais («en dur») dans notre programme et aucune interaction avec l'utilisateur n'était proposée.
Nous allons maintenant créer une ***application interactive*** qui proposera dans un premier temps à un utilisateur de ***créer un Patient*** en mode console.
Comme vous l'avez vu en UML, le ***flot de base de ce use case*** est le suivant :
(création d'un patient sans ascendant)

Flot de base

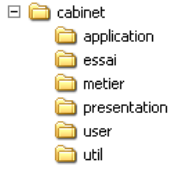
1. le système affiche la liste des informations à saisir,
2. l'utilisateur saisit les informations de la fiche patient et valide
3. le système vérifie la saisie et enregistre le nouveau patient, et affiche un message de prise en compte,
4. l'utilisateur choisit de quitter,
5. le système ferme le use case.

La mise en place d'une interactivité a une incidence sur l'architecture logicielle d'une application.

Il est bien sûr évident qu'il faut éviter d'écrire toutes les classes au même niveau et qu'il faut organiser son application en « package » (couche) afin de faciliter le développement (répartition des tâches) et la maintenance de l'application…

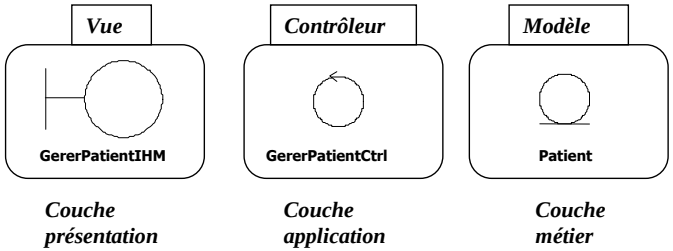
Choix d'une architecture logicielle :
Avant de se lancer dans la programmation d'une application, il est donc important de réfléchir à son architecture.
Le concept d'architecture *multi-tiers* (ou *n-tiers*) propose de découper une application en plusieurs ***couches logiques*** spécialisées chacune dans une fonction précise :
➔ ***couche présentation :*** présentation des informations à l'utilisateur, interface de saisie (IHM)
➔ ***couche application (navigation ou contrôle) :*** gestion du parcours de l'utilisateur entre les différentes parties de l'application.
➔ ***couche métier :*** implémentation des traitements directement liés au métier
➔ ***couche de persistance métier :*** implémentation des mécanismes permettant de sauvegarder les objets manipulés dans l'application sur un support de persistance et de gérer les accès à ces données.

La dénomination de ces différentes couches ne vous est pas inconnue puisque avant de commencer à implémenter le projet `cabinetMedical`, nous avions déjà organisé notre application en package afin de respecter « le découpage en couches issues de la conception » (cf TP1)



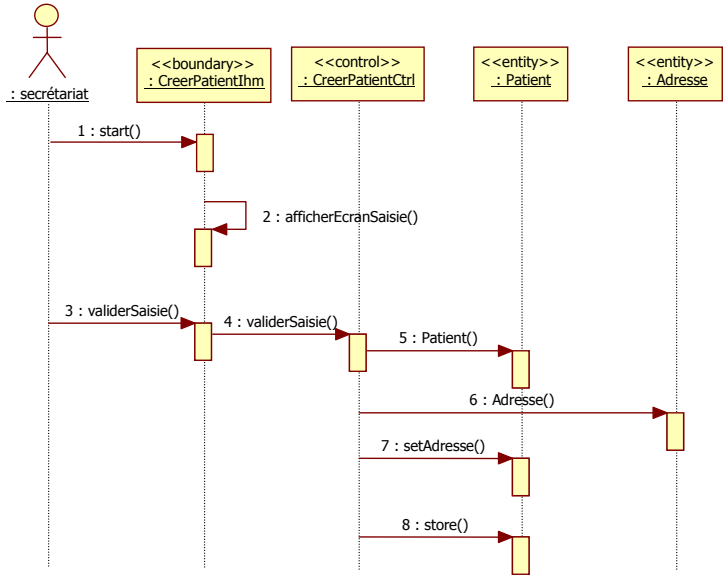
Architecture logicielle du projet `cabinetMedical`

Il est important lors de la mise en place d'une architecture de ne pas « réinventer la roue », mais de s'appuyer sur des patterns (modèles de programmations) robustes et simples qui ont fait leur preuve.
Pour écrire notre application, nous utiliserons le **pattern MVC** (**M**odèle **V**ue **C**ontrôleur) qui vous a été présenté en cours d'UML.



Dans le diagramme, nous identifions les 3 composants d'un modèle MVC :

- le **Modèle** (classe `Patient`) qui correspond à la classe ***métier*** qui contient la logique de l'application.
- la **Vue** (classe `GererPatientIHM`) qui permet de mettre en place une (re)***présentation*** visuelle de l'application à l'écran (pour aujourd'hui ce sera un mode console). Elle affiche des informations sur le modèle.
- le **Controleur** (classe `GererPatientCtrl`) qui est en fait le cœur de l'***application*** en assurant son côté événementiel. Il est d'ailleurs représenté au centre du diagramme de séquence.



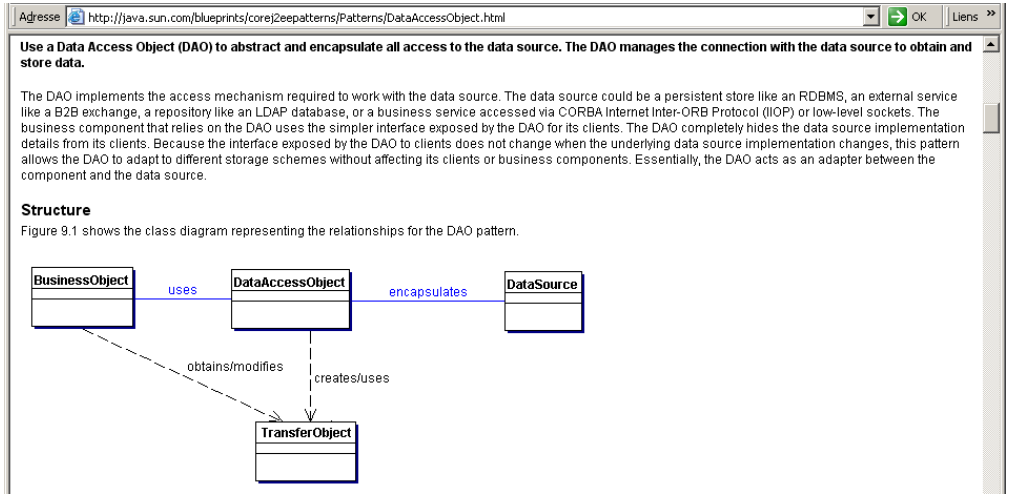
Persistance des objets métiers à l'aide du pattern DAO (Data Access Object)

Le pattern DAO est sans doute le modèle de conception le plus utilisé dans le monde de la persistance

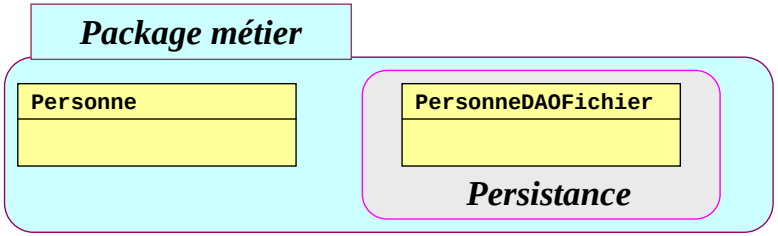
→ Une **classe de type `DAO`** doit proposer des méthodes pour implémenter le **`CRUD`**.
Cet acronyme (**`C.R.U.D.`**) désigne les **4 opérations de base de la persistance** :

- **`C`***`reate`* (ou `store`) pour créer une nouvelle entité dans le support de persistance
- **`R`***`etrieve`* (ou `find` ou `load`) pour retrouver une ou plusieurs entités dans le support de persistance
- **`U`***`pdate`* pour modifier une des entités du support de persistance
- **`D`***`elete`* (ou `remove`) pour supprimer une entité du support de persistance

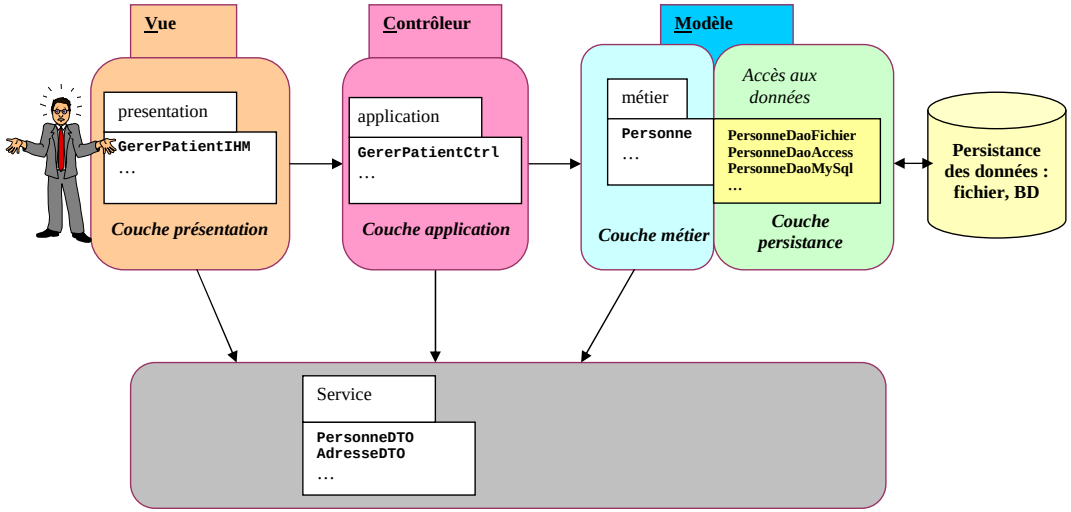
Pour chacune de ces opérations, on peut trouver ***plusieurs variantes de signatures des méthodes correspondantes*** dans les DAOs (c-a-d des méthodes surchargées)



Dans le cadre de notre application (architecture centralisée), nous développerons les classes DAO directement dans le paquetage ***metier***. Le DAO devra porter le même nom que la classe métier.



A Retenir sur le pattern DAO :
Le pattern DAO permet de s'abstraire de la façon dont les données sont stockées.
Il facilite une éventuelle "interchangeabilité" du support de persistance en encapsulant les accès aux données à des endroits distincts de l'architecture (plutôt que ceux-ci soient dispersés).
Afin de séparer le plus possible les entrées-sorties des classes métiers, il utilise des couches :
→ ***la couche DAO d'accès aux données : « cache » la connexion*** à tout support de persistance
C'est dans cette couche que se trouvent réellement les instructions d'accès aux données.
Les méthodes proposées par cette couche sont celles du CRUD (ajout, lecture, modification et suppression)
Avantage : La couche métier est complètement découplée du support de persistance.
Dans le cadre des bases de données, le programmeur utilisant le pattern DAO n'a plus besoin de connaître le langage SQL pour pouvoir manipuler des données, il lui suffit d'appeler la méthode du CRUD adéquate.



Les DTO (Data Transfer Object)

Un DTO est donc une classe qui possède les mêmes attributs que ceux de la classe métier.
→ Cette classe est composée uniquement **d'attributs et de getteurs/setteurs** (respect du principe d'encapsulation) et des 3 méthodes classiques **`toString`** et **`equals, hashcode`** (pour pouvoir manipuler les Collections)
→ Il n'y aura ***aucune méthode de traitement*** dans le `DTO` (dans le cas de `PersonneDTO`, la classe ne contiendra pas la méthode calculée `getAge`).
→ Par contre, comme dans la classe métier correspondante, le DTO redéfinira également les 3 méthodes héritées de la classe `Object`, à savoir **`toString`** et **`equals, hashcode`** (pour pouvoir manipuler les Collections)
→ Un **`DTO`** est un objet de **Transfert** : c'est une ***copie de l'objet métier***.
Il n'y aura donc aucune vérification sur la nature des données transportées par le DTO (pas d'appel à la méthode `verifierNir`, ni de test sur l'existence ou non d'une `Adresse`: ce sont les objets métiers qui se chargent de vérifier ces règles). Un DTO "n'est donc pas risqué" ce qui signifie que le ***DTO ne lèvera aucune exception !!!*** (c-a-d qu'aucune `CabinetMedicalException` ne doit apparaître dans les DTO de notre application)
→ Un DTO ne sera pas enregistré dans le support de persistance (c'est l'objet métier qui l'est).
Par contre un DTO, comme son nom l'indique est un objet de ***T**ransfert* : il peut être amené à aller sur le réseau (dans le cas d'une architecture multi-tiers). C'est pour cela que **le DTO doit implémenter `Serializable`.**
→ Les `DTO` sont donc des classes qui traversent toutes les couches du modèles **`MVC`** .

- **`Modele` → `DTO` → `Vue` (via `Ctrl`) :** Lors d'une recherche sur un objet métier, le DTO transporte les données du Modèle que la Vue doit afficher.
- **`Vue` → `DTO` → `Modele` (via `Ctrl`) :** Lors d'une création d'un objet métier, le DTO contient les données du formulaire de la Vue et les transportent jusqu'au Modèle.

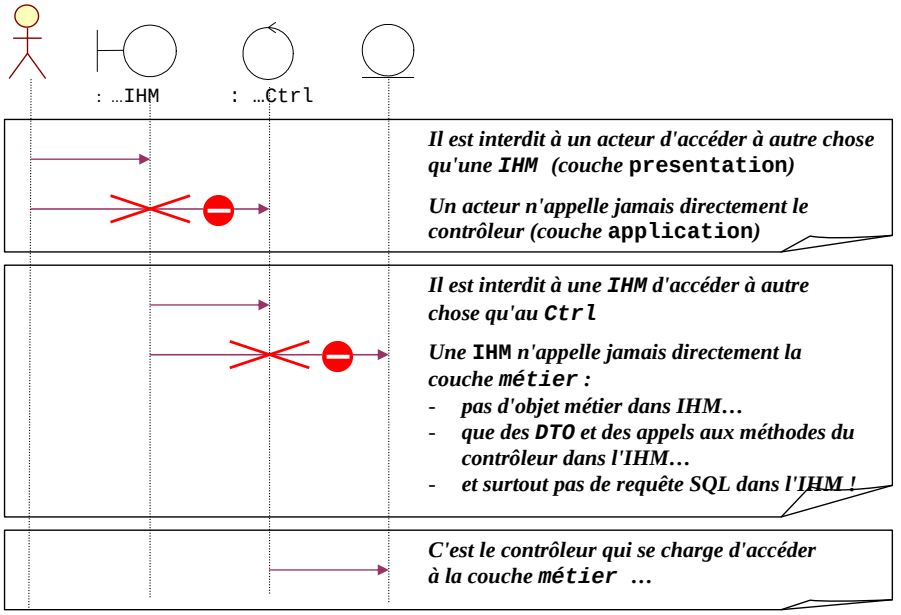
Ainsi, le DTO permet de sécuriser le support de persistance : *en manipulant un DTO, on ne risque pas de modifier les données réelles, puisqu'on travaille* ***juste sur des copies****.*
→ Afin que toutes les couches (**`M,V,C`**) puissent utiliser les DTO, on place habituellement les DTO dans une ***couche transversale accessible à toutes les couches appelée également « couche de service »*** et représentée dans notre application par le package `com.iut.cabinet.user`.

Les `Helper` pour favoriser la conversion des objets métiers ↔ objets `DTO`

Pour pouvoir utiliser les `DTO` dans notre application, il nous manque une dernière classe à écrire, c'est la **classe `Helper`**. La classe `Helper` permet de transmettre les données du DTO à l'objet métier et inversement.
Elle est donc composée initialement de deux méthodes de type :

```
public static MaClasseDTO toMaClasseDTO (MaClasse unObjetDeMaClasse) ;
public static MaClasse toMaClasse (MaClasseDTO unObjetDeMaClasseDTO) ;
```

➢ Il vous faut bien garder à l'esprit les ***quelques "règles" que nous venons de voir*** : chaque couche a une responsabilité. La mise en place de l'application sous forme de couches logicielles permet d'isoler les comportements/actions de chacune des couches. On ne "saute" pas de couches…



Correction TD JAVA n°4: Collection et Généricité

Exercice 1 : Manipulation des Collections : Comparaison et tri des objets …

1. Dans un premier temps, écrire un programme **`TestTriMontagnes.java`** correspondant au jeu d'essai suivant :

```
--- Des Montagnes ---
Mont Blanc        --> 4807 m d'altitude
Puy de Sancy      --> 1886 m d'altitude
Puy de Dôme       --> 1464 m d'altitude
Pic du Midi       --> 2877 m d'altitude
Pic d'Aneto       --> 3404 m d'altitude
Pic du Canigou    --> 2784 m d'altitude
```

```
import java.util.ArrayList;

public class TestTriMontagnes {

      public static void main(String[] args) {
      new TestTriMontagnes();
      }

      public TestTriMontagnes(){

      // Instanciation d'une collection lesMontagnes
      ArrayList<Montagne> lesMontagnes = new ArrayList<Montagne>();

      //Ajout d'objets de type Montagne à la collection
      Montagne mont1=new Montagne("Mont Blanc",4807);
      lesMontagnes.add(mont1);
      lesMontagnes.add(new Montagne("Puy de Sancy",1886));
      lesMontagnes.add(new Montagne("Puy de Dôme",1464));
      lesMontagnes.add(new Montagne("Pic du Midi",2877));
      lesMontagnes.add(new Montagne("Pic d'Aneto",3404));
      lesMontagnes.add(new Montagne("Pic du Canigou",2784));

      //Affichage du contenu de la collection
      System.out.println("--- Des Montagnes ---");
       for(Montagne uneMontagne : lesMontagnes)
            {System.out.println(uneMontagne);}
      }
}
```

Remarque : l'affichage aurait aussi bien pu faire avec un `while` …
En n'oubliant pas de rajouter en haut du fichier : `import java.util.Iterator;`

```
// Affichage avec un while
Iterator <Montagne> it = lesMontagnes.iterator();
 while(it.hasNext()){
   Montagne uneMontagne = it.next();
   System.out.println(uneMontagne);
   }
```

… mais code un peu plus long qu'avec un `for` (`foreach` qui de plus, n'a pas besoin de import…)

Remarque : on pourrait aussi écrire directement : `System.out.println(lesMontagnes);`
Puisque la méthode `toString` est redefinie pour les objets de type `Montagne` stockés dans la collection `lesMontagnes`.
Par contre affichage entre `[ ]` et les élements sont séparés par des virgules => `toString` des collections formate comme cela ! ! !

```
--- Des Montagnes ---
[Mont Blanc      --> 4807 m d'altitude, Puy de Sancy     --> 1886 m d'altitude, P
```


2. Que pensez-vous des déclarations suivantes ?

`ArrayList<Montagne> lesMontagnes = new ArrayList<Montagne>();`	☒ Correct ☐ Erreur de compilation
`List<Montagne> lesMontagnes = new ArrayList<Montagne>();.`	☒ Correct ☐ Erreur de compilation
`Collection<Montagne> lesMontagnes = new ArrayList<Montagne>();`	☒ Correct ☐ Erreur de compilation

Les 3 déclarations sont correctes, n'oubliez cependant pas d'utiliser le bon import…

```
import java.util.ArrayList;
ArrayList<Montagne> lesMontagnes = new ArrayList<Montagne>();

import java.util.List;
List<Montagne> lesMontagnes = new ArrayList<Montagne>();

import java.util.Collection;
Collection<Montagne> lesMontagnes = new ArrayList<Montagne>();
```

`List` et `Collection` sont des interfaces (classes « toutes » abstraites non instanciables), mais on peut bien sûr déclarer un objet de type List en utilisant une classe concrète comme `ArrayList` (ou `LinkedList` ou `Vector`…)
Intérêt : lorsque l'on passe des collections en paramètre dans des fonctions, mieux vaut passer la classe « la plus haute dans la hiérarchie » c-a-d `Collection` : cela laisse ensuite libre choix à l'utilisateur d'utiliser la classe concrète qu'il veut (`ArrayList`, `LinkedList` et pourquoi pas des classes concrètes héritées de l »interface `Set`).
Ici, quelle que soit l'instanciation que l'on choisit on travaille sur la **classe concrète `ArrayList`**, puisque l'appel au constructeur est bien dans tous les cas : `new ArrayList<Montagne>();`

Et de celles-là ? (révision sur le polymorphisme)

`List<Montagne> lesMontagnes = new List<Montagne>();.` ***Erreur*** car `List` est une ***interface*** donc ***non instanciable*** (c-a-d pas d'appel à `new` possible du type ~~`new List`~~)	☐ Correct ☒ Erreur de compilation
`ArrayList<Montagne> lesMontagnes = new List<Montagne>();` ***Erreur*** idem `List` est une ***interface*** donc ***non instanciable*** (c-a-d pas d'appel à `new` possible du type ~~`new List`~~)	☐ Correct ☒ Erreur de compilation

3. Tri de la collection suivant un seul critère.

3.1 Retrouvez dans le cours, l'instruction permettant d'effectuer un tri sur les collections, adaptez là à notre collection (transparent n°32).

`Collections.sort(lesMontagnes)`

Quand on lit la documentation de ArrayList, on ne trouve aucune méthode relative aux tris.
Remonter la hiérarchie d'héritage ne nous aide pas non plus.
Heureusement qu'il y a la classe Collections qui propose des méthodes statiques pour faire des manipulations sur les collections…=> méthode statique d'où l'appel à la méthode indexé par le nom de la classe.

3.2 Modifier la classe `Montagne` pour pouvoir utiliser l'instruction de la question 3.1 et effectuer à l'aide de cette instruction un ***tri par nom des montagnes*** partir de la méthode `sort` proposée dans le cours (et extraite de l'API) :

```
public static <T extends Comparable<? super T>> void sort(List<T> list)
```

…. Il faut bien sûr travailler avec le cours ….

La documentation nous dit que nous ne pourrons passer `ArrayList<Montagne>` à la méthode `sort` que si la classe `Montagne` implémente l'interface `Comparable.`
Rappel interface Comparable

```
public interface Comparable <T>{
     // retourne un entier négatif, zéro ou un entier positif
     //suivant que l'objet (de la classe) est inférieur,
     // égal ou supérieur à l 'objet spécifié en paramètre
     int compareTo(T o) ;

}
```

L 'objet qui exécute la méthode `compareTo` (ici `Montagne`) doit déterminer si la `Montagne` reçue doit être placée plus haut, plus bas ou au même emplacement dans la liste, suivant la valeur renvoyée par `compareTo`.

```
public class Montagne  implements Comparable<Montagne>{

……

// Comparaison par rapport au champ nom de la montagne …
     public int compareTo(Montagne m) {

          return nom.compareTo(m.getNom());
// il suffit de déléguer aux objets String (type de nom)
// puisque la classe String contient déjà une méthode compareTo implémentée

}

}
```

La méthode sort va transmettre une Montagne à compareTo pour voir comment son NOM se compare par rapport au NOM de celle qui a été invoquée la méthode compareTo.

L 'objet qui exécute la méthode `compareTo` (ici `Montagne`) doit déterminer si la `Montagne` reçue doit être placée plus haut, plus bas ou au même emplacement dans la liste, suivant la valeur renvoyée par `compareTo`

Remarque : De toutes façons on est **obligé d'utiliser `compareTo`** car on ne peut pas écrire : ~~`nom < m.getNom()`~~ car la surchage d'opérateurs n'existe pas en Java (enfin pas encore...), contrairement au C++ ...
… on ne peut pas non plus utiliser la méthode `equals` car elle ne nous donne pas d'informations sur l'ordre des éléments.

```
--------- compareTo sur les String ---------
  ---> si String nom1 vaut : aaa
  ---> si String  nom2 vaut : bbb
 nom1.compareTo(nom2) vaut : -1
 nom2.compareTo(nom1) vaut : 1
  --------- compareTo sur les Integer ---------
  ---> si Integer int1 vaut : 7
  ---> si Integer  int2 vaut : 14
 int1.compareTo(int2) vaut : -1
 int2.compareTo(int1) vaut : 1
```

on ne peut pas utiliser `compareTo` avec des `int` …
`int int1; int int2;` ~~`int1.compareTo(int2)`~~ ne marche pas car `compareTo` doit s'appliquer sur un objet ! ! !

4. Tri de la collection suivant deux critères : *Tri des montagnes par nom et par hauteur :*

4.1 Ecrire une classe **`ComparateurNom`** qui implémente l'interface **`Comparator`** sur des objets de type **`Montagne et`** permet de trier ces objets (`Montagne`) suivant leur **`nom`**.
Bon à savoir pour le TP … Il est d'usage d'implémenter les comparateurs comme des classes dans le même fichier des classes qui les utilisent. Nous implémenterons donc **`ComparateurNom`** sera implémentée dans le fichier **`TestTriMontagnes`**

```
public class TestTriMontagnes {
          ////////////////////////////////////////////////
          // … Votre code déjà écrit
          // concernant la classe TestTriMontagnes …
          ////////////////////////////////////////////////////
} //fin classe TestTriMontagne
```

En dehors de la classe, mais dans le même fichier ⇒ à la volée !

```
class ComparateurNom implements Comparator<Montagne>
          {
          public int compare(Montagne mont1, Montagne mont2)
               {
               return mont1.getNom().compareTo(mont2.getNom());
// ré-utilisons le compareTo des String !!!
// on est obligé de passer par getNom() car nom est private !!!
// et on est quand même dans une autre classe,  même si elle est interne …
               }
} //fin classe Comparateur Nom
```

Pourquoi l'implémentation du comparateur se fait-elle généralement en tant que classe interne ?

En effet, nous avons besoin de la classe **`ComparateurNom`** pour trier la collection qui est une variable propre à la classe **`TestTriMontagnes`**. C'est pourquoi les comparateurs seront en général implémentés « à la volée » c-a-d à l'endroit et au moment où on en a besoin.

En effet, même s'il est possible d'écrire plusieurs classes par fichiers (avec une seule classe publique qui doit porter le même nom que le fichier), il vaut mieux écrire des classes internes pour les comparateurs, car ces comparateurs sont propres) la classe en cours, et non pas besoin d'^étre connus en dehors… …

4.2 Ecrire les instructions permettant de trier la liste de montagnes par nom `(lesMontagnes)` avec un appel à la méthode `sort` surchargée à 2 paramètres.
2 étapes :

→ Créer une instance de la classe interne Comparator (c-a-d notre **`ComparateurNom)`**

→ appeler la méthode sort surchargée en lui transmettant à la fois la liste des montagnes et notre comparateur (instance de Comparator)

```
//Tri des montagnes suivant le nom avec la méthode sort avec 2
paramètres
     ComparateurNom  compareNom = new ComparateurNom();
     Collections.sort(lesMontagnes,compareNom);
```

4.3 De la même manière, mettre en place une classe **`ComparateurHauteur`** qui permet de trier les montagnes suivant leur altitude, et écrire le tri à l'aide de la méthode sort et d'un objet de cette classe.

```
class ComparateurHauteur implements Comparator<Montagne>
     {
          public int compare(Montagne mont1, Montagne mont2)
          {
               return (mont1.getHauteur()-mont2.getHauteur());
// ordre croissant des hauteurs (du plus bas sommet au plus haut)
               //return new
Integer(mont1.getHauteur()).compareTo(mont2.getHauteur());
               //  attention, on pourrait utiliser compareTo
               // ... mais pas directement sur un int !!!
               // car compareTo doit s'appliquer sur un objet...
               // l'auto-boxing est ensuite automatique pour le
paramètre ...
               // écriture un peu lourde quand même...
          }
     }
```

Remarques :

→ Si on avait voulu ***l'ordre décroissant*** des sommets du plus haut au plus petit …

```
     return (mont2.getHauteur()-mont1.getHauteur());
```

→ **Si on avait voulu utiliser compareTo (bien que à mon avis non adpaté ici)**

```
     return new
     Integer(mont1.getHauteur()).compareTo(mont2.getHauteur());
     //  attention, on pourrait utiliser compareTo
     // ... mais pas directement sur un int !!!
     // car compareTo doit s'appliquer sur un objet...
     // l'auto-boxing est ensuite automatique pour le paramètre ...
     // écriture un peu lourde quand même...
```