TP à réaliser en <u>binôme</u>. Le type de questions est très proche du genre de questions posées lors de l'examen de fin de semestre (sauf les tests unitaires JUnit).

En informatique, un traitement typique (en l'absence de base de données) consiste à

1. Lire des données depuis un fichier (texte ou binaire).
2. Effectuer un traitement sur ces données (les trier, les modifier, supprimer ou rajouter des données, voir simplement vérifier la validité d'un sous-ensemble de données).
3. Ecrire les nouvelles données dans un fichier.

<u>Travail à faire :</u> Vous devez écrire un programme en Java qui permet de gérer une liste de contacts téléphoniques que peut avoir une entreprise :

- Lecture d'un fichier contenant les contacts et sauvegarde des contacts courants.
- Gestion d'une liste de contacts (ajout, suppression, recherche, s'assurer de l'absence de doublon, tri selon certains critères).
- Code professionnel : commentaires Javadoc, lancement d'exception lors de la détection d'anomalie et tests unitaires des fonctionnalités via JUnit.

<u>Exercice 1.</u> L'héritage, les exceptions et les collections

- Un contact est défini par un **<u>nom</u>**, un **<u>prénom</u>**, un **<u>numéro de téléphone</u>** (ce sont 3 chaînes de caractères) et une **<u>date</u>** de création du contact (de type java.time.LocalDate). Il possède également un **<u>identifiant</u>** généré automatiquement qui est un entier : la première instance d'un contact créée portera le numéro 1, la deuxième le numéro 2,…

Il existe des contacts particuliers : il s'agit des **clients** et des **fournisseurs** de l'entreprise. Un **<u>client</u>** a une donnée supplémentaire par rapport à un contact traditionnel : le **<u>chiffre d'affaires</u>** réalisé avec ce client (un réel positif). Les **<u>fournisseurs</u>** de l'entreprise sont caractérisés par deux données supplémentaires. Il y a un **<u>montant total</u>** acheté auprès de ce fournisseur (un réel positif) et un **<u>rabais moyen</u>** effectué sur le prix des produits (il s'agit d'un pourcentage, donc un réel entre 0 et 100).

On souhaite gérer une **<u>liste de contacts</u>** grâce à une classe nommée ***<u>ListeContacts</u>*** sur laquelle on pourra effectuer les opérations suivantes :

1. **<u>ajouter</u>** un contact dans la liste.
→On passera en paramètre à notre méthode le contact à ajouter. On interdira d'avoir 2 contacts avec le même nom, le même prénom et même numéro de téléphone : dans ce cas on génèrera une exception.

```
public void ajouter(Contact c)
```

2. **<u>supprimer</u>** un contact en fournissant son identifiant.
→On passera, en paramètre, l'identifiant du contact à supprimer. Si le contact n'existe pas, on génèrera une exception.

```
public void supprimer(int id)
```

3. **<u>afficher la liste des contacts</u>** (en mode texte).
→On renverra toutes les caractéristiques de nos contacts ainsi que leur type.

```
public String toString()
```

4. **afficher la liste des contacts de type *Client*** (en mode texte).
→On affichera toutes les caractéristiques de nos contacts clients en utilisant un itérateur.

```
public void afficherClients()
```

5. **afficher la liste des contacts de type *Fournisseur*** (en mode texte).
→On affichera toutes les caractéristiques de nos contacts fournisseurs en utilisant une boucle **for** optimisée (syntaxe Java 5).

```
public void afficherFournisseurs()
```

- **Type des contacts**
  - Ecrire les classes : **Contact**, **Client**, **Fournisseur** et valider les méthodes de ces classes via des tests unitaires JUnit.
  - On mettra un accent sur la réutilisation de code afin de ne pas réécrire des services similaires dans des classes sœurs.
  - Le type de chaque classe (interface, classe abstraite ou classe concrète) devra être précisé dans les commentaires Javadoc.
  - Proposer des améliorations sur la gestion des anomalies : par exemple le lancement d'une exception si le numéro de téléphone n'est pas valide.
- **Gestion d'un nombre arbitraire de contact :** la classe **ListeContacts**
  - Choix des attributs et constructeurs. **Attention**, pour représenter les contacts vous devez utiliser une collection Java et non un tableau Java. Le choix du type concret de la collection devra être justifié.
  - Méthodes publiques de la classe : ajouter, supprimer, toString, afficherClients, afficherFournisseurs ; on pourra aussi proposer une méthode de recherche, qui devra être efficace, i.e. en O(log(tailleListe)). Valider ces méthodes via des tests unitaires JUnit.
  - Gestion des exceptions. On pourra aussi valider ces exceptions via des tests unitaires JUnit : https://github.com/junit-team/junit4/wiki/Exception-testing
- Les 4 classes précédentes devront être placées dans le package *contact*.
- Proposer 2 types de tri des contacts (tri maintenu dans l'affichage suivant): le tri lexicographique des contacts par nom+prénom, et le tri des contacts selon leur ancienneté (du plus ancien au plus récent). Valider vos méthodes avec des tests unitaires.

Exercice 2. La persistance des objets

Vous devez proposer une solution pour rendre persistant la liste des contacts de l'entreprise, via un fichier binaire. Vous devrez en particulier :

- Proposer une méthode de sauvegarde d'un objet de type `ListeContacts` sous la forme d'un fichier *binaire* : `public void sauvegarder(String nomDeFichierBinaire) ;`
- et proposer un constructeur pour la classe ListeContacts, `public ListeContacts (String nomDeFichierBinaire)`, qui prend en compte le nom d'un fichier binaire pour pouvoir construire un objet de type `ListeContacts`.
- Des exceptions seront jetées dans les cas problématiques (e.g. si le fichier spécifié est introuvable).
- Tester les deux méthodes précédentes (Tests unitaires JUnit).

Attention : le mécanisme de génération des identifiants des contacts ne devra pas être impacté par la sauvegarde puis la recréation via un constructeur.