Manipuler et trier des Collections

Pour cet exercice, vous pouvez vous aider de la documentation en ligne de la classe « ArrayList »

a) Créer un programme Java qui crée une collection de pays (Classe Country avec un attribut name) puis alimenter cette collection avec quelques valeurs et afficher la taille de la collection
Exemple de résultat à obtenir :

```
La collection créée contient 4 pays !
```

b) Compléter le programme pour afficher le contenu de la collection.

Exemple de résultat à obtenir :

```
La collection créée contient 4 pays !
France
Allemagne
USA
Chine
```

c) Trouver une méthode pour vider la collection et modifier votre programme afin d'afficher un message lorsqu'elle est vide et afficher le contenu lorsqu'elle n'est pas vide.

Exemple de résultat à obtenir :

```
Liste vide
```

d) Après avoir de nouveau alimenté votre liste de pays, modifiez le nom d'un pays et affichez de nouveau la liste des pays.

Remarque : Il y a plusieurs méthodes pour manipuler des éléments dans une collection de type ArrayList (ajouter, modifier ou supprimer). Choisissez celle que vous voulez dans la documentation Oracle.

e) Triez votre collection et réaffichez la liste des pays.

Dans la classe « ArrayList », une méthode « sort » est nécessaire pour trier notre collection.

- **Dans la documentation, trouvez-vous cette méthode ?**

Elle n'existe pas. En revanche, allez voir la documentation de la classe utilitaire « Collections ».

- **Désormais, trouvez-vous cette méthode « sort » ?**

Une « ArrayList » implémente l'interface Collection, elle est donc une collection. Nous pouvons donc trier par ordre alphabétique croissant ou décroissant une Arraylist avec la syntaxe suivante « **Collections.sort(uneArrayList)** » pour trier une ArrayList **de *String***.

```
List<String> countries = new ArrayList<>();
countries.add("FRANCE");
countries.add("BELGIQUE");
Collections.sort(countries);
```

Si nous souhaitons trier notre collection dans l'ordre inverse :

```
Collections.sort(countries, Collections.reverseOrder());
```

Tester le code ci-dessous sur une ArrayList de String, puis sur une Arraylist de Country.

- **Que se passe-t-il ?**

Pour utiliser la méthode « sort » sur une collection d'objets autre que String, il nous faut utiliser un comparateur.

Créer une classe **CountryComparator** qui implémente l'interface ***Comparator*** que vous utiliserez dans votre méthode « **sort** » afin de trier vos **Country** par ordre alphabétique croissant ou décroissant.

```
Collections.sort(countries, new CountryComparator());
```

f) Triez votre collection du plus petit mot au plus grand mot. Réaffichez ensuite la liste des pays.

Exemple de résultat à obtenir :

USA
Chine
France
Allemagne

Comprendre les exceptions

1. **Lever une exception**
2. **Capturer ou propager**
    a. **Propagation de l'exception**
    b. **Traitement de l'exception**
3. **Compléments sur les exceptions**

L'un des éléments donnant sa puissance et sa robustesse au langage Java est sa gestion avancée des exceptions.
Une application gérant correctement ses exceptions doit alors pouvoir réagir à toutes les situations y compris les cas imprévus par le développeur. Dans ce sens, une erreur d'exécution peut toujours être rattrapée et le programme pourra continuer sans sortie brusque. Dans la pratique, on utilise la gestion des exceptions dans deux cas :

- Lorsque l'on s'attend à une erreur précise (par exemple, l'absence de fichier lors d'une lecture d'un disque)

- Pour protéger l'intégrité du programme : une erreur imprévue peut se produire et une partie du programme peut échouer. Dans ce cas, on peut décider de l'action à effectuer pour la suite du programme : sortie ou nouvel essai...

1. Lever une exception

Une exception est un événement particulier apparaissant au cours de l'exécution (la plupart du temps une erreur) .

Une exception est un objet dérivant de la classe *java.lang.Exception*.

Il existe toutes sortes d'exceptions dont certaines font partie de l'API Java standard comme les *ArithmeticException* . D'autres comme une hypothétique *'VoitureSansPlaqueException'* qui sont développées spécifiquement pour une application donnée.

Exemple d'exception :

```
public class VoitureSansPlaqueException extends Exception {

    public String toString() {
        return "Cette voiture ne possède pas de plaque d'immatriculation";
    }
}
```

Nous verrons plus loin à quoi sert la méthode *toString().* Il est important de noter que dans la majorité des cas, vous n'aurez pas à implémenter de nouvelles exceptions mais simplement à utiliser celles déjà existantes.

Les exceptions personnelles servent à créer des types d'erreur spécifiques qui permettront une granulosité plus fine à la gestion des événements. Dans le cas où on ne cherche pas à déterminer la cause précise de l'erreur mais simplement à détecter un problème quelconque pour afficher un message générique, nous pouvons directement utiliser les objets de type Exception .

Les exceptions sont dites **levées** ou **envoyées** par le mot-clé ***throw***.

Exemple :

La méthode *getPlaque* prend en argument un objet *Voiture* et renvoie un objet de type *PlaqueImmatriculation*. Si la voiture ne possède pas de plaque, le programme considère qu'une erreur applicative est commise et lève alors une *VoitureSansPlaqueException* :

```
public PlaqueImmatriculation getPlaque(Voiture v) throws VoitureSansPlaqueException
{
    if (v.plaque==null){
        throw new VoitureSansPlaqueException();
    }
    else{
        return v.plaque;
    }
}
```

Le mot-clé **new** instancie l'exception et le mot-clé **throw** lance l'exception qui va alors être traitée ou propagée.

Remarque : Attention à ne pas confondre le mot clé **throw** qui sert à lever une exception avec le mot clé **throws** qui sert à propager l'exception.

2. Capturer ou propager

Lorsqu'une exception est levée, il y a deux possibilités pour le programme :

- Il propage l'exception (mot-clé **throws**) ;
    - Au rugby, le joueur passe le ballon à un équipier.

- Il capture l'exception et la traite (mot-clés **try/catch**) ;
    - Au rugby, le joueur marque l'essai.

Détaillons ces deux processus : **(a et b)**

a) Propagation de l'exception

Considérons l'exemple de la méthode *getPlaque* :

```
public PlaqueImmatriculation getPlaque(Voiture v) throws VoitureSansPlaqueException
{

    if (v.plaque == null) {
        throw new VoitureSansPlaqueException();
    } else {
        return v.plaque;
    }
}
```

Nous voyons que dans certains cas, cette méthode peut lever une exception. Cependant, elle ne la traite pas.

Elle n'affiche pas de message d'erreur et n'effectue aucun traitement particulier. Elle se contente de propager l'exception à la méthode appelante par le mot-clé **throws**.

Dans la déclaration de la méthode, "*throws VoitureSansPlaqueException*" signifie que la méthode est susceptible de propager l'exception *VoitureSansPlaqueException*.

Dans ce cas, ce sera à la méthode appelante de traiter l'exception ou alors de la propager à son tour. La propagation d'une exception de méthodes en méthodes peut être affichées par la méthode *printStackTrace()* de l'exception.

b) Traitement de l'exception

L'exception sera propagée de méthodes en méthodes tant qu'elle ne sera pas traitée.
Pour traiter une exception, il suffit d'utiliser les mots-clés *try* et *catch*:

```
try{
    getPlaque(v1);
    afficherPlaque();
}
catch(VoitureSansPlaqueException vspe){
    vspe.printStackTrace();
    afficherMessageErreur();
}
```

Le mot-clé ***try*** précise que les instructions contenues dans ces accolades peuvent lever des exceptions dont le type sera précisé en argument du *catch* associé.

Si la voiture *v1* ne possède pas de fournisseur, la méthode *getPlaque()* - comme nous l'avons vu- lèvera une exception de type *VoitureSansPlaqueException* et la propagera. A la ligne *getPlaque(v1)*, l'exception est alors capturée. La ligne *afficherPlaque()* n'est pas exécutée et le programme passe alors aux instructions du ***catch*** correspondant.

Résumé de ce mécanisme

Lorsqu'une exception est levée, le programme arrête son exécution et ne reprendra que dans un *catch()* adéquat de l'exception. La méthode qui a levé l'exception va alors la traiter immédiatement ou va la propager à la méthode appelante qui à son tour va traiter l'exception ou la propager. La seule façon de stopper ce parcours sera de traiter l'exception (*catch*). Il faut voir ce phénomène comme des passes au rugby: l'exception est le ballon et il est transmis de joueurs en joueurs jusqu'à celui qui va 'traiter l'exception' et marquer l'essai.

3. Compléments sur les exceptions

Maintenant que nous maîtrisons les mécanismes fondamentaux des exceptions, nous allons détailler certaines méthodes de programmation.

Les méthode *toString()* et *printStackTrace()* de la classe *Exception*

Cette méthode renvoie la chaîne de caractère décrivant l'erreur. Il est souvent judicieux de surcharger cette méthode à l'implémentation d'une classe fille de Exception. La classe Exception quant à elle ne contient pas de message par défaut, il peut être utile de construire un objet Exception avec un String décrivant le problème.

Exemple :

```
public void test() throws Exception{
    if ( – pbm de division par zéro –){
        throw new Exception("Division par Zéro");
    }
    if (– pbm de nombre trop petit –){
        throw new Exception("Nombre trop petit");
    }
}
```

L'instruction suivante :

```
try{
    test();
}
catch(Exception e){
    System.out.println(e);
}
```

Affichera "Division par zéro" ou "Nombre trop petit" selon le cas.

Mieux encore : L'instruction ***e.printStackTrace()*** ou '*e*' est l'exception affichera toujours les messages d'erreurs mais donnera le parcours complet de l'exception du moment où elle a été levée jusqu'à celui où elle a été capturée. Cette méthode de la classe Exception est très utile au débogage.

Capture sélective

Il est possible de réaliser des filtres d'exceptions dans des cas complexes. Par exemple, admettons que la méthode *getPlaque* vue précédemment puisse lever une *VoitureSansPlaqueException* mais également tout autre type d'exception. Dans ce cas, le code sera le suivant :

```
try{
    getPlaque(v1);
}
catch( VoitureSansPlaqueException vspe){
    vspe.printStackTrace();
}
catch( Exception e){
    e.printStackTrace();
}
```

Nous effectuons ainsi une capture sélective avec granulosité décroissante. Bien entendu, aucune exception ne pourra passer outre un *catch(Exception)* puisque toutes les exceptions dérivent de la classe ***Exception***.

Si nous voulons réaliser une action qu'il y ait une exception ou non, il faut utiliser le mot-clé ***finally***

```
try{
    getPlaque(v1);
}
catch( VoitureSansPlaqueException vspe){
    vspe.printStackTrace();
}
catch( Exception e){
    e.printStackTrace();
}
finally{
    System.out.println("Traitement terminé");  //ce code est exécuté qu'il y ait exception ou non
}
```

Remarque sur les déclarations de variables

Comme pour les boucles itératives, les variables locales définies dans un *try* ou un *catch* ne sont connues que dans ces zones.

À vous de jouer ! Gérer les exceptions

g) Dans l'exercice précédent, vous avez normalement utilisé une méthode pour afficher votre liste. Si la liste est vide, un message s'affiche.

Pour l'exercice, considérons que l'affichage d'une liste vide est un flot anormal d'exécution.

Créez une **Exception** personnalisée **ListEmptyException**. Vous propagerez celle-ci lorsque la liste comporte une taille de 0.

Capturer cette exception lors de l'exécution de cette méthode, et afficher un message personnalisé. Utilisez **throw, throws, try and catch**.

Exo Exceptions 1 : Division par 0

1) Écrire un programme qui effectue une division par zéro et ne contient aucun traitement d'exception. Que se passe-t-il? Pourquoi? Quel est le type de l'exception générée.

2) Cette fois, réécrire le programme pour capturer l'exception
3) Le modifier pour afficher un message d'erreur explicite
4) Cette fois, le programme corrige lui-même et remplace la division par zéro par une division par 1.

Remarquons ainsi que lever une exception ne signifie forcement l'affichage d'un message d'erreur suivi de l'arrêt du programme mais qu'il peut y avoir poursuite normale de l'exécution.

Exo Exceptions 2 : L'âge du capitaine

1) Écrire une méthode *getAgeCap()* qui demande l'âge du capitaine. Cet âge doit être compris entre 18 et 65 ans et doit être un entier sous peine de lever une *AgeCapException*. Vous implémenterez cette exception pour qu'elle renvoie une description explicite du type "*[proposition] ans n'est pas un âge valide*". Le programme devra également être en mesure de capturer tout type d'exception autre que *AgeCapException*.
2) Dans un premier temps, la méthode *getAgeCap()* propagera l'exception à la méthode appelante qui la traitera.
3) Modifier le programme pour que ce soit la méthode *getAgeCap()* qui traite l'exception.
4) Modifier encore le programme pour que *getAgeCap()* traite l'exception mais lève une seconde exception de type *Exception* pour signaler à la méthode appelante qu'une erreur s'est produite et que cette dernière comptabilise le nombre d'essais infructueux et l'affiche. Le programme demandera l'âge du capitaine en boucle infinie.

Exo Exceptions 3 : Saisir un mot de passe

Dans les failles de sécurité réseau, on trouve très souvent les problèmes de dépassement. Par exemple, sur certaines anciennes versions de *telnet*, un login ou un mot de passe de plus d'un mega-octet faisait "planter" le programme et on obtenait alors un accès *root* au système. Ce programme va gérer ce type de problème en séparant les exceptions pour une meilleure gestion.

1) Écrire un programme stand-alone qui demande en boucle un nom d'utilisateur (login) et un mot de passe (pwd) jusqu'à recevoir un login/pwd correct.

Le seul utilisateur référencé sera *scott / tiger* (à mettre en constante dans la classe principale).

2) Implémenter les exceptions suivantes :

* ***WrongLoginException*** qui se produit lorsque l'utilisateur saisit un login inexistant

* ***WrongPwdException*** lorsque le mot de passe est erroné

* ***WrongInputLength*** lorsque le ***login*** où le ***pwd*** saisi dépasse 10 caractères.

3) Implémenter de façon à utiliser ces exceptions de façon judicieuse et avec la granulosité la plus fine.

Exo Exceptions 4

Soit le programme EssaiException.java suivant :

```
import java.util.Scanner;

public class EssaiException {
    public static void main(String[] args) {
        int a, b, res;
        Scanner clavier = new Scanner(System.in);
        a = clavier.nextInt();
        b = clavier.nextInt();
        res = a / b;
        System.out.println("le résultat de " + a + " divisé par " + b + " est " + res);
        System.out.println("Fin du programme");
    }
}
```

Si vous êtes dans un état normal, vous devez être persuadé que notre programme recèle une faille importante car nous divisons sans vérifier que le diviseur n'est pas nul. Néanmoins, lancez le programme en saisissant une valeur nulle pour b.

a) **Le programme s'est-il exécuté correctement ?**
b) **Le message "Fin du programme" est-il apparu ?**
c) **Quelle exception a été levée par la machine Java ?**

Nous allons maintenant faire en sorte que le programme ne se termine pas aussi brutalement, et nous renseigne un peu plus.

Pour cela, nous allons mettre en place un bloc try/catch afin d'attraper l'exception levée précédemment :

```
        try {
            res = a / b;
            System.out.println("le résultat de " + a + " divisé par " + b + " est " + res);
        } catch (ArithmeticException e) {
            System.out.println("oop ! un problème dans la division ");
            System.out.println("le message officiel est " + e.getMessage());
        }
        System.out.println("Fin du programme");
```

Relancez le programme en saisissant une valeur nulle pour b.

d) **Le programme a t-il affiché qu'il y avait un problème dans la division ?**
e) **Le message "Fin du programme" est-il apparu ?**
f) **Quel est le message d'erreur officiel correspondant à une telle exception ?**

A la suite de l'instruction catch vous allez maintenant rajouter le bloc **finally** suivant

```
finally {
    System.out.println("le bloc finally sera toujours exécuté") ;
    System.out.println("et c'est là que l'on fermera par exemple les fichiers") ;
}
```

et vous relancerez le programme en saisissant encore une valeur nulle pour b.

g) **Le bloc finally a t-il été exécuté ?**

Mettez en commentaire le bloc catch et relancerez le programme en saisissant encore une valeur nulle pour b.

h) **Le bloc finally a t-il été exécuté ?**
i) **L'exception a t-elle été traitée ?**

Pour terminer, vous allez maintenant relancer le programme en saisissant une lettre à la place d'un nombre

j) **Que se passe t-il ?**
k) **Quelle exception a été lancée ?**

Corriger le problème en traitant l'exception lancée et en intégrant les lectures au clavier dans le bloc try.

Exo Exceptions 5

Soit le programme **oubliStupide.java** suivant :

```
public class OubliStupide {
    public static void main(String[] args) {
        int[] tab = null;
        System.out.println(tab[2]);
    }
}
```

Où l'on utilise un tableau sans l'avoir créé. Mettez en place le traitement de l'exception lancée lorsque le programme s'exécute. Attention, je ne vous demande pas de corriger le programme en rajoutant :

```
int[] tab = new int[10];
```

Exo Exceptions 6

Soit le programme adresseIP.java suivant :

```
public class AdresseIP {

    public static void main(String[] args) {
        AdresseIP adr = new AdresseIP(267, 277, 1929, 10);

    }

    private int[] octet;

    public AdresseIP(int o1, int o2, int o3, int o4) {
        octet = new int[]{o1, o2, o3, o4};
    }
```

```
    public String toString() {
        return octet[0] + "." + octet[1] + "." + octet[2] + "." + octet[3];
    }
}
```

Vous allez mettre en place un traitement par exception pour éviter que l'on puisse construire un adresse IP avec des nombres farfelus. Pour cela, vous allez définir votre propre classe d'exception dérivant de la classe de base **Exception**

```
public class ExceptionAdrIP extends Exception {
    public ExceptionAdrIP(String s) {
        super(s);
    }
}
```

Puis vous allez modifier la classe **AdresseIP** de la manière suivante :

```
public class AdresseIP {

    private int[] octet;

    public AdresseIP(int o1, int o2, int o3, int o4) throws ExceptionAdrIP {

        if (o1 < 0 || o1 > 255) {
            throw new ExceptionAdrIP("valeur incorrecte pour le premier octet");
        }

        if (o2 < 0 || o2 > 255) {
//            A vous de compléter suivant le même modèle
        }

        octet = new int[]{o1, o2, o3, o4};
    }

    public String toString() {
        return octet[0] + "." + octet[1] + "." + octet[2] + "." + octet[3];
    }
}
```

Et enfin, vous mettrez en place dans la fonction main le traitement des exceptions de type **ExceptionAdrIP** afin que lors de la levée d'une telle exception, l'erreur soit récupérée et le message indiquant l'octet faux soit affiché.