I) Préambule sur le déroulement des exercices d'application du cours

Vous trouverez ci-dessous des exercices d'applications qui correspondent à des applications du cours.

Je vous prie de faire dans l'ordre :

1) Installez l'IDE de votre choix exemple Eclipse, IntelliJ, NetBeans etc.…

2) Lire le cours **Support-Formation-Java-SE-Fondamentaux.pdf** en détail en testant les exemples de code dessus (il est important de consacrer à ce cours 3 à 5 heures afin d'en extraire toute la quintessence).

3) Enfin terminer en faisant dans l'ordre les exercices de l'**étape 1 - les bases du langage Java** puis ceux de l'**étape 2 Collections/Comparable/Comparator**, puis lire le préambule sur les Design Pattern, puis les exercices de l'**étape 3 (Design Pattern Singleton et Factory)** et l'exercice de l'**étape 4 – JDBC (Java DataBase Connectivity**.

II) Etape 1 - les bases du langage Java

Le but de cette étape est de vérifier que les notions de base de programmation orienté objet sont bien comprises. On va juste faire quelques petits exercices d'introduction à java.

- Récupérer le projet Maven **maven-intro-poo-java** ou **app-intro-poo-java** et completez les methodes **testCompareChaines, testFactorielItterative, testFactorielRecursive, testNombreMagique, testSortMyIntegerArray, testSortMyStringArray et testProcessAnimal** de la classe **com.cours.main.Main**. Il s'agira de d'implementer les méthodes ci-dessus à travers les sections **Exercice test Pair/Impair, Exercice comparaison chaine de caractères, Exercice nombre Magique, Exercice de trie de tableau, Exercice Factorielle, Exercice Polymorphisme et Exercice de manipulation de classes avec la classe Animal** ci-dessous en utilisant la classe **Calculation**.

  Pour les methodes **testCompareChaines, testFactorielItterative, testFactorielRecursive, testNombreMagique, testSortMyArray** nous utiliserons les méthodes **compareChaines, factorielItterative, factorielRecursive, nombreMagique, initMyIntegerArray, sortMyIntegerArray, initMyStringArray** et **sortMyStringArray** de la classe **Calculation**.

  Pour la méthode **testProcessAnimal** nous utiliserons l'interface **IProcessAnimal** et ce sera à vous d'implementer les méthodes de sa classe d'implémentation **ProcessAnimal** (les méthodes **getAnimals, loadAnimalsManually, loadAnimalsFile, calculMoyennePoidsAnimaux, calculEcartTypePoidsAnimaux, sortAnimalsById, sortAnimalsByName, sortAnimalsByWeight, sortAnimalsByColor, generateFileByName et generateFileByWeight**).

1) Test Pair/Impair

Compléter la méthode **test**V**erifyParite** qui est un programme Java qui lit un nombre avec le clavier et indique s'il est positif, négatif ou s'il vaut zéro et s'il est pair ou impair.

Exemple d'exécution:

Entrez un nombre entier: 5
Le nombre est positif et impair

Entrez un nombre entier: -4
Le nombre est négatif et pair

Entrez un nombre entier: 0
Le nombre est nul.

Vous utiliserez la méthode **verifyParite** (bien entendu il faudra que vous l'implémentez) de la classe **Calculation** (**verifyParite** renvois 0 si le nombre est égale à 0, 1 s'il est pair et positif, -1 s'il est négatif et paire, 2 s'il est impair et positif et -2 s'il est négatif et impair).

2) Factorielle

Compléter les méthodes **testFactorielItterative** et **testFactorielRecursive** qui sont des programmes qui calculent et affichent le factoriel d'un petit nombre entiers positifs (entre 0 et 12) entrés au clavier de deux manières :

- De manière itérative avec la formule itérative :

  n! = 1 * 2 * 3 * ... * n
- De manière récursive avec la formule récursive définissant n! en fonction de (n-1)! :

  0! (factorielle de zéro) = 1

  Pour tout entier n>0, n! = n * (n-1)!

Vous utiliserez les méthodes **factorielItterative** et **factorielRecursive** de la classe **Calculation** (il faudra que vous les implémentez). Pour des raisons de simplicité de code nous allons uniquement calculer les factorielles de nombre en 0 et 12. En effet si on dépasse 12 alors on sort la capacité maximale de stockage du type Integer qui est **2 147 483 647**.

3) Nombre Magique

Compléter la méthode **testNombreMagique** qui est un programme qui remplie un tableau à une dimension de 100 éléments avec des valeurs aléatoires entre 1 et 100.

Afficher les éléments de ce tableau.

Calculer le nombre magique de votre tableau. Le nombre magique correspond au maximum des éléments du tableau auquel on ajoute son minimum et enfin on divise le résultat par 2.

Afficher le nombre magique de ce tableau d'éléments aléatoires.

Vous utiliserez la méthode **initMyArray** (il faudra que vous l'implémentiez) de la classe **Calculation** pour remplir votre tableau de 100 éléments aléatoires et la methode **nombreMagique** (il faudra que vous l'implémentiez) de la classe **Calculation** pour calculer le nombre magique de votre tableau d'éléments aléatoires entre 1 et 100.

4) Comparaison chaine de caractères

Compléter la méthode **testCompareChaines** qui est un programme qui permet à l'utilisateur de rentrer deux chaines de caractères au clavier.

Le programme affichera :

« Les deux chaînes sont identiques » Si les chaines de caractère sont égales.
« La première chaine est supérieur à la deuxième. » Si la première chaine est supérieure à la deuxième.
« La deuxième chaine est supérieur à la première ». Si la deuxième chaine est supérieur chaine est supérieur à la première.

Vous utiliserez la méthode **compareChaines** (bien entendu il faudra que vous l'implémentez) de la classe **Calculation** (**compareChaines** renvois 0 si les deux chaines de caractères sont identiques, 1 si la première chaine est supérieur à la seconde chaine (en terme de code ASCII) et -1 si la première chaine est inférieur à la seconde chaine (en terme de code ASCII)). Ce sera à vous de concevoir un algorithme simple de comparaison de deux chaines de caractères.

Il vous est interdit pour cet exercice et uniquement pour cet exercice d'utiliser la méthode compareTo native de Java, ce sera à vous d'implementer une methode avec un algorithme simple qui compare deux chaines de caractères en terme de code ASCII.

Le caractere « a » a pour code ASCII 97, «b » a pour code ASCII 98 et « c » a pour code ASCII 99 et ainsi de suite. Pour plus d'informations on vous inscite vivement à consulter l'article suivant https://www.commentcamarche.com/contents/93-code-ascii.

Le but de cet exercice est de vous aider à comprendre comment la JVM compare de manière optimisée deux chaines de caractères. Dans les autres exercices on pourra utiliser compareTo native de Java qui est surement beaucoup plus optimisé que n'importe quel algorithme de comparaison que vous implementerez.

5) Trie de tableau d'entier

Compléter la méthode **testSortMyIntegerArray** qui est un programme qui remplit un tableau à une dimension de 100 éléments avec des valeurs aléatoires entre 1 et 100.

Afficher les éléments de ce tableau.

Réaliser un programme de trie par ordre croissant des éléments ce tableau par un algorithme de tri simple et rudimentaire de votre choix (tri par sélection, tri par extraction, le trie par insertion, le trie par fusion etc…).

Réafficher les éléments de ce tableau pour vérifier que votre trie a bien fonctionné.

Vous utiliserez la méthode **initMyIntegerArray** (il faudra que vous l'implémentiez) de la classe **Calculation** pour remplir votre tableau de 100 éléments aléatoires et la méthode **sortMyIntegerArray** (il faudra que vous l'implémentiez) de la classe **Calculation** et l'appeler dans **testSortMyIntegerArray**.

Il vous est interdit pour cet exercice et uniquement pour cet exercice d'utiliser les méthodes Arrays.sort et Collections.sort de Java. Le but de cet exercice est de vous aider à comprendre comment la JVM trie de manière optimisée des données. Par la suite par exemple dans l'exercice « Collections/Comparable/Comparator » on pourra utiliser Arrays.sort et Collections.sort de Java qui sont surement beaucoup plus optimisés que n'importe quel algorithme de trie que vous implémenterez.

6) Trie de tableau de chaine de caractères

Compléter la méthode **testSortMyStringArray** qui est un programme qui remplit un tableau de chaine de caractères à une dimension de 100 éléments avec des mots aléatoires de 10 caractères.

Afficher les éléments de ce tableau.

Réaliser un programme de trie par ordre croissant des éléments ce tableau par un algorithme de tri simple et rudimentaire de votre choix (tri par sélection, tri par extraction, le trie par insertion, le trie par fusion etc…).

Réafficher les éléments de ce tableau de chaines de caractères pour vérifier que votre trie a bien fonctionné.

Vous utiliserez la méthode **initMyStringArray** (il faudra que vous l'implémentiez) de la classe **Calculation** pour remplir votre tableau des mots aléatoires de 10 caractères et la méthode **sortMyStringArray** (il faudra que vous l'implémentiez) de la classe **Calculation** et l'appeler dans **testSortMyStringArray**.

Il vous est interdit pour cet exercice et uniquement pour cet exercice d'utiliser les méthodes Arrays.sort et Collections.sort de Java. Le but de cet exercice est de vous aider à comprendre comment la JVM trie de manière optimisée des données. Par la suite par exemple dans l'exercice « Collections/Comparable/Comparator » on pourra utiliser Arrays.sort et Collections.sort de Java qui sont surement beaucoup plus optimisés que n'importe quel algorithme de trie que vous implémenterez.

7) Moyenne et Ecart type

Compléter les méthodes **Calculation.calculMoyenne** et **Calculation.calculEcartType** qui sont des méthodes qui calculent la moyenne et l'écart type de la liste de poids passé en paramètre.

8) Heritage Polymorphisme

Compléter la classe Mammifere avec les attributs :

- Un attribut nomMammifere de type String.
- Une méthode description qui affiche une description complète du mammifère (Exemple : « Ceci est un Mammifère, son nom est **nomMammifere**. »).

Créer les classes Mouton, Chat, Lion et Requin qui héritent de la classe Mammifère dans le package com.cours.polymorphe.entities.

La classe Mouton va afficher à travers sa méthode description : « Ceci est un Mouton, son nom est **monMouton**. ».

La classe Chat va afficher à travers sa méthode description : « Ceci est un Chat, son nom est **monChat**. ».

La classe Lion va afficher à travers sa méthode description : « Ceci est un Lion, son nom est **monLion**. »

La classe Requin va afficher à travers sa méthode description : « Ceci est un Requin, son nom est **monRequin**. »

Completer la méthode **ProcessPolyMorphe.initArrayMammmiferes** en initialisant 5 mammifères, 5 moutons, 5 chats, 5 lions et 5 requins dans un tableau.

Adapter la signature de méthode **ProcessPolyMorphe.dislayOneMammmifere** qui affichera la description quel que soit le mammifère passé en paramètre.

Compléter la méthode **ProcessPolyMorphe.dislayAllMammmiferes** qui affichera les descriptions de tous les mammifères passées en paramètres.

9) Manipulation des fichiers avec la classe Animal

Je vous rappelle que cet exercice est indépendant de l'exercice précédant sur le Polymorphisme, nous utiliserons uniquement l'entité **Animal** que vous allez définir ci-dessous.

- Compléter la classe « **Animal** » avec les attributs suivants :

« idAnimal » de type « Integer », « nom » de type « String », « poids » de type « double », « dateNaissance » de type « java.util.Date », « couleur » de type «String», « nombrePattes » de type « int », « estCarnivore » de type «boolean» (booléen).
La classe « Animal » a aussi les méthodes suivantes :

« deplacer () » qui affiche : L'animal « nom de l'animal (nom) » se déplace avec « nombre de pattes (nombrePattes) » pattes
.
« description () » qui affiche : L'animal « nom de l'animal (nom) » est née le « date de naissance de l'animal (dateNaissance) », il pèse « poids du l'animal (poids) », il est de couleur « couleur de l'animal (couleur) », il a « nombre de pattes (nombrePattes) » pattes, il est carnivore (ou n'est pas carnivore)

Remarque : « carnivore » ou « n'est pas carnivore » suivant la valeur de « estCarnivore».

- Ajouter les méthodes **toString**, **equals** et **hashCode** (equals et hashCode seront basé sur idAnimal). A quoi servent ces méthodes ? (mettez les réponses dans un fichier **readMe.txt**)
- Completer la methode **loadAnimalsManually** de la classe **ProcessAnimal** qui va charger un ensemble de 11 objets de type **Animal** dans une **List** de type **Animal** (**List<Animal>**). Vous chargerez par exemple un Lion, un Eléphant, un Tigre, un Requin, un Chat, un Mouton, une Chèvre, une Poule, un Porc, un Singe et un Giraffe.
- Nous allons par la suite charger des animaux se trouvant dans le fichier texte **animaux.txt**. Vous trouverez en pièces jointes le fichier **animaux.txt**.

Compléter la méthode **loadAnimalsFile** de la classe **ProcessAnimal** qui va récupérer les animaux du fichier **animaux.txt** et les charger dans un **List** de type « **Animal** » (**List<Animal>**).
Le contenu du fichier **animaux.txt** est :

1,Lion, 200, 23/12/1988, Maron, 4, true
2,Elephant, 700, 23/12/1978, Noir, 4, true
3,Tigre, 150, 23/12/1998,Blanc, 4, true
4,Requin, 10,23/12/1978,Blanc,0,true
5,Chat, 5,23/12/2000,Noir,4,true
6,Mouton,25,23/12/2001,Blanc,4,false
7,Chevre,35,23/12/2012,Noir,4,false
8,Poule,1,23/12/2009,Maron,2,false

9,Porc,20,23/12/2003,Blanche, 4, true
10,Singe,25,23/12/2004,Noir,4,false
11,Giraffe,175,23/12/2013,Maron et Noir, 4, false

- Compléter la méthode **calculMoyennePoidsAnimaux** de la classe **ProcessAnimal** qui calculera la moyenne des poids des animaux du fichier **animaux.txt**. Compléter ensuite la méthode **calculEcartTypePoidsAnimaux** de la classe **ProcessAnimal** qui calculera l'écart type des poids des animaux du fichier **animaux.txt** (on pourra utiliser **Calculation.calculMoyenne** et **Calculation.calculEcartType**).
- Implémenter les méthodes **sortAnimalsById, sortAnimalsByName, sortAnimalsByWeight** et **sortAnimalsByColor** qui sont respectivement le trie d'animaux par l'attribut « idAnimal » de l'animal, par l'attribut « nom » de l'animal, par l'attribut « poids » de l'animal et par l'attribut « couleur » de l'animal. On pourra s'inspirer de la méthode **sort** qui a été implémenté dans l'exercice de trie de tableau (nous pourront adapter la comparaison des nombre de l'exercice en comparaison de chaine de caractères).
- Implémenter les méthodes **generateFileByName** et **generateFileByWeight** pour générer les fichiers **animauxParNom.txt** qui contiendra la liste des animaux du fichier **animaux.txt** à qui on a appliqué un trie par nom et le fichier **animauxParPoids.txt** à qui on a appliqué le trie par poids. Il est important que les fichiers **animauxParNom.txt** et **animauxParPoids.txt** soit exactement au même format de données que le fichier **animaux.txt** donc attention au format de date de naissances.

Indications :

La moyenne se calcule de la manière suivante :

$$\bar{x}=\frac{1}{n}\sum_{i=1}^{n} x_i$$

L'écart type se calcule de la manière :

$$\sigma_X := \sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i-\bar{x})^2} = \sqrt{\frac{1}{n}\left(\sum_{i=1}^{n} {x_i}^2\right) - \bar{x}^2}$$

Xi Étant le poids d'un animal.
$\bar{X}$ étant la moyenne des poids.
N étant le nombre d'animaux.

Vous n'avez pas besoins de spécifier le chemin du fichier **animaux.txt** dans la mesure où il est à la racine de votre projet. Le simple fait de mettre le nom du fichier vous permet de trouver le fichier texte.

III) Etape 2 Collections/Comparable/Comparator

Récupérer le projet Maven **maven-collections-comparator** ou **app-collections-comparator** puis :

1) Compléter la classe **com.cours.entities.Personne** avec les attributs suivants :
« idPersonne» de type « Integer », « prenom » de type « String », « nom » de type « String », « poids» de type « Double », « taille » de type « Double », « rue» de type « String », « ville» de type « String », « codePostal» de type « String ».

Méthodes :

« toString () » qui retourne toutes les informations sur les attributs de la classe « Personne ».
« equals () » qui retourne un boolean true si les deux personnes ont le même prenom et le même nom.
« hashCode () » qui retourne un hash par rapport au prenom et nom.
« getImc () » qui retourne un double qui représente l'indice de masse corporelle.
« isMaigre () » qui retourne un boolean représentant l'état de maigreur de la personne.
« isSurPoids () » qui retourne un boolean représentant l'état de surpoids de la personne.
« isObese () » qui retourne un boolean représentant l'état d'obésité de la personne.
« compareTo (Personne other) » qui retourne un entier par rapport à une comparaison d'une personne avec idPersonne.

Indications :

Vous trouverez sur le lien ci-dessous toutes les informations pour les calculs des IMC :

https://fr.wikipedia.org/wiki/Indice_de_masse_corporelle

Dans tout l'exercice on initialisera les données avec les données ci-dessous.

1, Maurice, Dupont, 100, 170, rue du paradis, Rouen, 76000, France
2, Martin, Marshall, 55, 150, rue de Nantes, Laval, 53000, France
3, Claire, Chazal, 65, 175, rue de Rennes, Laval, 53000, France
4, Celine, Dia, 87, 170, rue Diderot, Paris, 75000, France
5, Remy, Cheval, 63, 140, rue du paradis, Nantes, 44000, France
6, Nicolas, Dutrou, 79, 155, rue Appert, Nantes, 44000, France
7, Marie, Claire, 65, 166, rue du paradis, Rouen, 76000, France
8, Nathalie, Sage, 89, 190, rue Appert, Rouen, 76000, France
9, Jean, Dujardin, 75, 140, rue des sorciers, Havre, 76800, France
10, Michel, Leclerc, 89, 190, rue du bonneur, Havre, 76800, France
11, Julien, Marshall, 60, 145, rue de Nantes, Laval, 53000, France
12, Julien, Claire, 78, 170, rue du Paradis, Paris, 75000, France

13, Jacques, Dupont, 63, 179, rue des Passeurs, Paris, 75000, France
14, Charles, Hallyday, 100, 200, rue des Feugrais, Rouen, 76000, France
15, Serge, Lama, 102, 195, rue des Heureux, Nantes, 44000, France
16, Vincent, Thomas, 81, 183, rue de Paris, Rennes, 35000, France
17, Eric, Dummat, 61, 155, rue de Versaille, Paris, 75000, France
18, Nicolas, Samuel, 64, 145, rue de Saint Louis, Laval, 53000, France
19, Rémy, Guerry, 79, 191, rue des Sages, Lyon, 69000, France
20, Nicolas, Drapeau, 56, 166, rue Mitterrand, Limoges, 87000, France

2) Utiliser la classe **ProcessCollections** et compléter la méthode **initArrayPersonnes** qui consiste à créer un tableau avec les 20 personnes ci-dessus, puis compléter **displayArrayPersonnes** qui consiste à afficher toutes les personnes.

   - Faire la même chose avec **initListPersonnes** qui consiste à créer une liste avec les 20 personnes ci-dessus, puis compléter **displayListPersonnes** qui consiste à afficher toutes les personnes.

   - Faire la même chose avec **initMapPersonnes** qui consiste à créer un dictionnaire (avec comme clés idPersonne et comme valeur Classe Personne) avec les 20 personnes ci-dessus, puis compléter **displayMapPersonnes** qui consiste à afficher toutes les personnes.

3) Compléter la classe **PersonneComparator** pour être en mesure de faire un tri des personnes d'une collections par **idPersonne, prenom et nom** (ascendante et descendante à chaque fois).

4) Utiliser la classe **ProcessComparator** et le comparateur **PersonneComparator** pour réaliser les méthodes **sortByIdAsc, sortByIdDesc, sortByPrenomAsc, sortByPrenomDesc, sortByNomAsc et sortByNomDesc** qui réaliseront les tries ascendants et descendants.

5) Faire des appels des classes **ProcessCollections** et **ProcessComparator** dans la classe **Main** pour vérifier que vos implémentations fonctionnent parfaitement.

IV) Préambule sur les Design Patterns

Un Design Pattern est une solution à un problème récurrent dans la conception d'applications orientées objet. Un patron de conception décrit alors la solution éprouvée pour résoudre ce problème d'architecture de logiciel. L'utilisation des Design Patterns offre de nombreux avantages. Tout d'abord, cela permet de répondre à un problème de conception grâce à une solution éprouvée et validée par des experts. Ainsi, on gagne en rapidité et en qualité de conception ce qui diminue également les coûts.
De plus, les Design Patterns sont réutilisables et permettent de mettre en avant les bonnes pratiques de conception.

Les Design Patterns sont représentés par :

- **Nom** : qui permet de l'identifier clairement
- **Problématique** : description du problème auquel il répond
- **Solution** : description de la solution souvent accompagnée d'un schéma UML
- **Conséquences** : les avantages et les inconvénients de cette solution

Les patrons de conception sont classés en trois catégories :

- **Création** : ils permettent d'instancier et de configurer des classes et des objets.
- **Structure** : ils permettent d'organiser les classes d'une application.
- **Comportement** : ils permettent d'organiser les objets pour qu'ils collaborent entre eux.

Pour plus de détails consulter la partie **Quelques designs patterns** du cours **Support-Formation-Java-Introduction.pdf** et le site : https://refactoring.guru/fr/design-patterns/

V) Etape 3 – Design Pattern Singleton + Factory

Avant de commencer cette partie lire la partie **Quelques designs patterns** du cours pdf **Support-Formation-Java-Introduction.pdf**.

Problématique : La direction des statistiques de la France voudrait créer un service Web pour donner la possibilité aux différentes agences de l'Etat (situées dans les différents départements) de récupérer une liste de personnes et les statistiques correspondantes (moyenne et écart type de poids et taille).

Comme les données sont enregistrées dans un fichier et que les calculs statistiques sont couteux (calcul moyenne des poids et tailles, calcul d'écart type des poids et tailles) alors il est indispensable de réaliser une architecture pour optimiser l'acquisition et le traitement des données.

Par exemple lire le fichier Csv ou Json ou Xml qu'une seule et unique fois et faire les calculs (moyenne et écart type poids et taille) et finalement stocker les données statistiques dans le Singleton.

La direction des statistiques de Paris préfère utiliser des fichiers CSV, Nantes préfère utiliser des fichiers XML et Laval préfère utiliser des fichiers JSON. Donc nous aurons besoins d'un système d'aiguillage qui nous permet de passer d'un format de données à un autre. Nous pourrons par exemple utiliser une factory.

Dans notre exemple, nous allons utiliser un fichier Json ou Xml avec seulement 20 entrées mais vous pouvez imaginer aisément, si nous faisions le calcul à l'échelle d'une ville (avec plus d'un millions de personnes donc fichiers avec plus d'un millions d'entrées), que le calcul serait très couteux et encore plus couteux si les employés des différentes villes accédaient aux fichiers en lançant le calcul (moyenne et écart type) plusieurs fois par jour donc un singleton sera plus que nécessaire pour ce genre de cas de situation.

Vous trouverez à la racine de votre projet Maven les fichiers CSV, Json et Xml (**personnesCsv.csv**, **personnesJson.json** et **personnesXml.xml**).

1) Récupérer le projet Maven **maven-design-pattern-singleton-factory** ou **app-design-pattern-singleton-factory**. La classe **Main** sera est notre classe de démarrage de notre application.

2) Récupérer le contenu de la classe **com.cours.entities.Personne** des précédents projets Maven.

3) Compléter la classe **CsvSingleton** qui sera un singleton (Holder de préférence) qui vont permettre de recuperer la liste des personnes, puis calculer la moyenne et l'écart-types des poids à travers le fichier Csv **personnesCsv.csv**. Ce singleton aura pour rôle de stocker dans la JVM la liste des personnes avec la moyenne et l'écart-types des poids et tailles qui correspondant respectivement à **personnesCsv.csv**.

4) Tester votre singleton dans votre **Main** afin d'être sûre qu'ils marchent bien.

5) Bonus (à faire si vous avez le temps) : compléter les classes **JsonSingleton** et **XmlSingleton** qui seront des singletons qui vont permettre de recuperer la liste des personnes, puis calculer la moyenne et l'écart-types des poids à travers les 2 formats de fichier Xml et Json (**personnesJson.json** et **personnesXml.xml**).

   Compléter la méthode **SingletonFactory.getFactory** pour permettre de passer d'un singleton à un autre en fonction du format de données choisit (CSV, JSON et XML) puis tester le passage d'un singleton à un autre la classe Main.

VI) Etape 4 – JDBC (Java DataBase Connectivity)

Avant de commencer cette partie lire la partie **Gestion base de données MySQL avec Java** du cours pdf **Support-Formation-Java-Introduction.pdf**. Ce sera vers la page 70 normalement.

1) Récupérer le projet Maven **maven-jdbc-drivermanager** ou **app-jdbc-drivermanager**. La classe **Main** sera est notre classe de démarrage de notre application.

2) Ouvrir le fichier **script_base_personnes.sql** à la racine de votre projet et créer votre base de données MYSQL.

3) Completer les methodes **createConnection, findAllPersonnes, findPersonneById, findPersonnesByPrenom, findPersonnesByNom, findPersonnesByCodePostal, findPersonnesByVille, createPersonne, updatePersonne et deletePersonne** de la classe **Main**.

4) Completer la classe **ConnectionSingleton** qui est un Singleton qui gerera une seule et unique connexion.

5) Completer la classe **PersonneDaoImpl** qui implementera l'interface **IPersonneDao** avec les contenu des methodes **createConnection, findAllPersonnes, findPersonneById, findPersonnesByPrenom, findPersonnesByNom, findPersonnesByCodePostal, findPersonnesByVille, createPersonne, updatePersonne et deletePersonne** de la classe **Main**.