Formation Java : Chargement et sélection de données

bernard

Table des matières

1 Contexte

Considérant les programmes qui composent un *système d'information*, on a vu que la modélisation des données qui nous intéressent spécifiquement (*entités* du domaine d'application) pouvaient être modélisées en Java par des *objets*, instances de *classes*.

Pour que ces données survivent à l'exécution des programmes, il faut qu'elles *persistent* sous la forme d'un stockage permanent, par exemple sous la forme de fichier.

Grâce aux possibilités d'abstractions fournies par le langage (héritage, implémentation d'*interfaces*), l'implémentation proprement dites (fichier local ou données récupérées à partir d'une URL par exemple), n'ont pas vraiment d'impact tant qu'on peut considérer la source comme un *flux* de données. On peut facilement lire l'intégralité des données et *construire* les objets au fur et à mesure en les stockant dans une structure de données, par exemple l'une des nombreuses *collections* de la bibliothèque standard Java.

On pourra ensuite utiliser ces données pour interroger le programme afin de sélectionner un certain sous-ensemble des données selon divers critères.

2 Exemple

On a repris l'exemple des villes caractérisées par :

— un nom

— une latitude

— une longitude

Comme une ville ne changera ni de nom ni de localisation, on en profitera pour définir une classe d'objets *immutables*, et bénéficier de la tranquillité d'esprit qui en découle, avec les attributs suivants :

```
1  private final String name;
2  private final double latitude;
3  private final double longitude;
```

On aura donc un constructeur :

```
1  public City(String name, double latitude, double longitude){
2      this.name= name;
3      this.latitude= latitude;
4      this.longitude= longitude;
5  }
```

Avec la possibilité de connaître la distance entre deux villes et entre un couple de coordonnées et une ville.

```
1  public double distanceTo(double latitude, double longitude){
2      double R= 6371e3;
3      double phi1=Math.toRadians(this.latitude);
4      double phi2=Math.toRadians(latitude);
5      double deltaPhi=Math.toRadians(latitude- this.latitude);
6      double deltaLambda= Math.toRadians(longitude- this.longitude);
7      double a= Math.sin(deltaPhi/2) * Math.sin(deltaPhi/2) +
8          Math.cos(phi1) * Math.cos(phi2) * Math.sin(deltaLambda/2) * Math.sin(deltaLambda/2);
9      double c= 2* Math.atan2(Math.sqrt(a), Math.sqrt(1-a));
```

```
10      return R * c;
11  }
```

Évidemment, pour la distance avec une ville, on réutiliser le code précédent :

```
1   public double distanceTo(Ville other){
2       return distanceTo(other.latitude, other.longitude);
3   }
```

Ne serait-ce que pour débugger, une méthode `toString()` de conversion en chaîne de caractères est toujours utile :

```
1   public String toString(){
2       return "Ville: {name: "+name+", latitude: "+latitude+", longitude: " + longitude+"}";
3   }
```

Pour trouver une ville la plus proche d'un point, on peut implémenter une méthode `closestTo` :

```
1   public static City closestTo(double latitude, double longitude
2                                , Collection<City> cities
3                                , double deltaD){
4       City res=null;
5       double minD= Double.POSITIVE_INFINITY;
6       for(City city : cities){
7           double currentD= city.distanceTo(latitude, longitude);
8           if((currentD < minD) && (currentD > deltaD) ){
9               minD= currentD;
10              res= city;
11          }
12      }
13      return res;
14  }
```

Cette implémentation suit le schéma classique d'une initialisation du résultat (`res`) à la valeur correspondant à un ensemble vide (`null`) puis d'une mise à jour de ce résultat pour tenir compte de chacun des éléments de l'ensemble à traiter. On remarquera que cet ensemble est pris en argument selon l'interface `Collection<City>` de façon à être le plus générique, et donc réutilisable, possible.

Le détail non évident est l'ajout d'une argument `deltaD`. Il servira à pouvoir réutiliser cette méthode pour trouver la ville la plus proche d'une ville donnée. Évidemment, la ville la plus proche est elle-même ! On pourrait passer en argument une copie de l'ensemble des villes à laquelle on aura retiré

la ville en question, mais il est aussi simple et plus performant de passer une valeur `deltaD` minimale telle qu'on en prendra pas en compte de ville située à une distance inférieure à `deltaD` du point de référence.

Dans le cas où l'on a pas l'utilité de cet argument (on cherche la ville la plus proche d'un point et non pas d'une ville), cet argument `deltaD` est à 0 et l'on peut *surcharger* la méthode pour cette valeur par défaut :

```
1  public static City closestTo(double latitude, double longitude
2                               , Collection<City> cities){
3      return clostestTo(latitude, longitude, cities, 0.);
4  }
```

Pour trouver une ville la plus proche d'une autre ville, on passe en argument les longitude et latitude de la ville considérée. Pour la valeur de `deltaD`, on doit se baser sur la précision des valeurs numériques et l'on peut utiliser la méthode statique Math.ulp :

```
1  public static City closestTo(City ref, Collection<Ville> cities){
2      return closestTo(ref.latitude, ref.longitude, cities, Math.ulp(1.));
3  }
```

Bien sûr, on voudra pouvoir désigner une ville non par la référence vers l'objet lui-même, mais par son nom. Il suffit pour cela de faire une fonction qui recherche une ville en fonction de son nom :

```
1  public static City findByName(String name, Collection<City> cities){
2      for(City city : cities){
3          if(city.name.equals(name)){
4              return city;
5          }
6      }
7      return null; // throws ?
8  }
```

On est dans un cas où l'on peut faire un retour prématuré (*early exit*) puisque dès qu'on a trouvé une ville avec le nom recherché, il n'est plus la peine de parcourir le reste de la collection.

Dans le cas où aucune ville n'a le nom recherché, on aurait à priori plusieurs résultats possibles :

— lancer une exception de type NoSuchElementException

— retourner une référence nulle

L'inconvénient de la référence nulle est qu'il est facile d'oublier de vérifier et traiter ce cas, ce qui produira une NullPointerException au moment où l'on essaiera d'utiliser la référence. Une solution plus moderne serait d'utiliser une valeur de retour de type =Optional<City>=, mais le gain est sujet à controverse.

3 Lecture des données

La lecture des données qui caractérisent une ville peut être faite trivialement, ligne par ligne, mais il faut prendre soin d'anticiper que le fichier ne sera pas parfait et qu'il y aura donc des données manquantes ou incorrectes. Dans ce cas, on passera juste la ligne correspondant à un enregistrement défectueux (on peut signaler l'erreur sur la sortie d'erreur `System.err`). Bien sûr, lors de la conversion en type numérique, il faudra prendre en compte le séparateur de chiffres décimaux qui est une virgule (`,`) dans le fichier alors que, par défaut, Java attend un point (`.`).

```
1  public static List<City> read(String citiesURL) throws IOException {
2      List<City> res= new ArrayList<City>();
3      URL url= new URL(citiesURL);
4      try(BufferedReader br = new BufferedReader(new InputStreamReader(url.openStream()))) {
5          br.readLine(); // skip header
6          for(String line = br.readLine(); line != null; line= br.readLine()){
7              String[] data = line.split(";");
8              if(data.length == 3){
9                  for(int i=1; i !=3; ++i){
10                     data[i]= data[i].replace(',','.');
11                 }
12                 try{
13                     res.add(new City(data[0], Double.parseDouble(data[1]), Double.parseDouble(data[2])));
14                 }catch(NumberFormatException e){
15                     System.err.println(e);
16                 }
17             }
18         }
19     }
20     return res;
21 }
```

4 Requêtes

4.1 Villes dans un rayon donné

On s'intéresse à un sous-ensemble des villes qui est à une distance strictement inférieure à un rayon donné par rapport à un centre donné. Une telle sélection peut facilement être implémentée :

```java
public static Set<City> closerThan(double latitude, double longitude, double dist, Collection<City> cities){
    Set<City> res= new HashSet<City>();
    for(City city : cities){
        if(city.distanceTo(latitude, longitude) < distance){
            res.add(city);
        }
    }
    return res;
}
```

La complexité algorithmique de cette requête est `O(n)` puisqu'il faut tester chacune des villes de l'ensemble. On pourrait faire mieux, mais ceci n'est pas trivial. En effet, on ne peut pas appliquer de recherche dichotomique puisqu'il n'y a pas de relation d'ordre totale selon laquelle trier nos villes.

On choisit d'utiliser une structure de donnée de type `Set` parce que l'on anticipe le besoin de faire (efficacement) des intersections. Sinon, on aurait pu retourner une collection implémentant par exemple l'interface `List` comme `ArrayList`.

4.2 Intersection de sous-ensembles de villes

Si l'on s'intéresse à une intersection de sous-ensembles de villes par exemple obtenus avec la méthode `closerThan`, il suffit d'utiliser la méthode retainAll de l'interface `Set`. **Attention : * l'appel de cette méthode modifie le sous-ensemble en question puisque c'est une *référence** sur la collection qui sera passée en paramètre. Il faut donc se poser la question du meilleur compromis entre :

tranquilité d'esprit on travaille sur une copie du sous-ensemble de façon à ne pas modifier celui-ci.

performance on évite le coût en temps et en mémoire de la copie et l'on modifie directement l'un des sous-ensemble passés en arguments et dont on veut calculer l'intersection.

5 Limitations

Bien qu'extrêmement limité, notre exemple montre déjà les limites d'une approche aussi simpliste de la manipulation de donnés

5.1 Validation nécessaire des données en lecture

Tout d'abord, on a pas de garanties sur le fait que les données stockées soient valides, ce qui oblige à valider à chaque lecture.

5.2 Limitation par la mémoire disponible

Le fait de charger l'ensemble des données en mémoire limite la quantité de données manipulables.

5.3 Performance de la sélection

Si la performance de nos fonctions de sélection (complexité algorithmique linéaire) est acceptable pour une utilisation interactive, elle ne permettrait pas de traiter très efficacement de façon automatique/répétée de grand nombres de requêtes.

5.4 Modifications ? (ajouts, altérations, suppressions)

Le principal problème de cette approche est qu'elle ne permet absolument pas de traiter de façon satisfaisante toutes modifications du jeu de données. Il serait inenvisageable de réécrire l'ensemble du fichier à chaque fois !

6 TODO Une solution : les bases de données