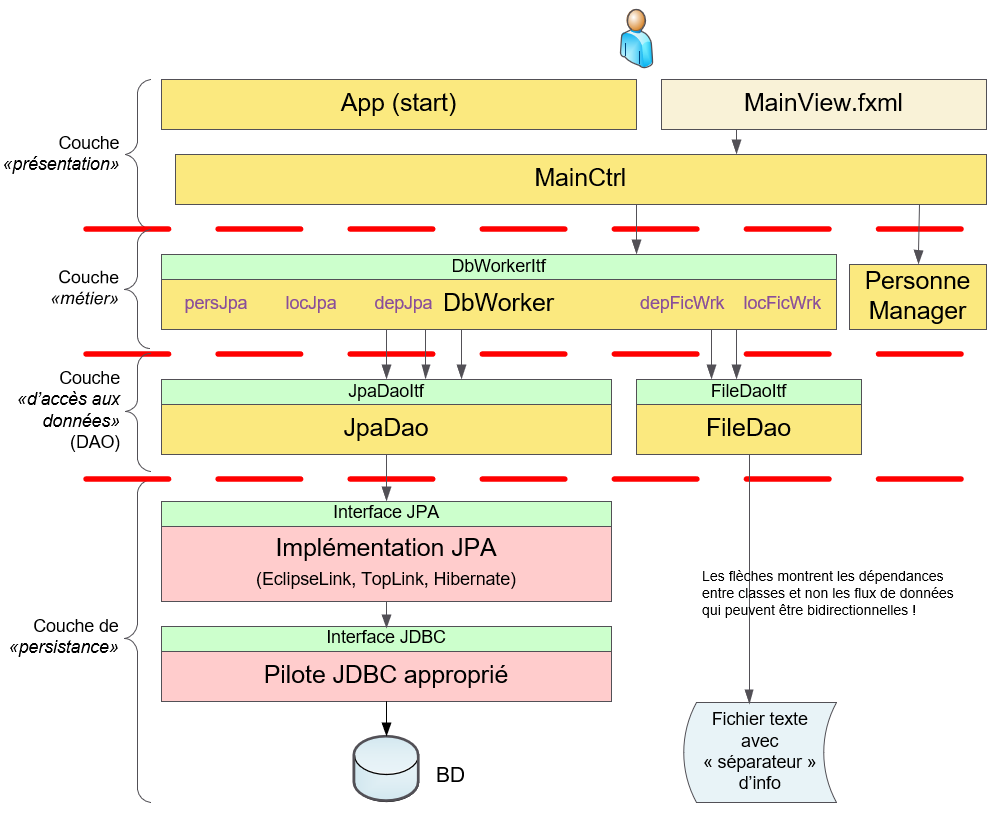
|  |  |
| --- | --- |
| Durée | 1 journée |
| Sujet | Utiliser JPA avec une sur-couche DAO |
| Objectifs | L’apprenant doit être capable, à la fin de la leçon :   * de créer dans un package *«****app.workers.dao****»* une couche d’accès aux données basée sur les workers JPAcréés préalablement ; * de créer dans *« app.workers.dao »* un worker générique *«****FileDao****»* pour lire des fichiers de type *« texte »*et extraire des données dans des bean ; * de créer dans *« app.workers.dao »* un worker générique *«****JpaDao****»* avec les méthodes déjà créées ; * de créer dans *« app.workers »* une classe *«****DbWorker****»* avec toutes les méthodes métier nécessaires et d’adapter aussi les classes *« Ctrl »* et *« Ihm »*; * de créer des méthodes de test pour toutes les méthodes cruciales. |

Le but de ce travail est de comprendre toute l’utilité d’une couche générique d’accès aux données (Data Access Layer ou *« DAL »,* aussi appelée communément *« DAO »*). Notre schéma des couches devient :



1. **Classe JpaDao**

En effet, dans l’exercice 7, nous avions eu besoin de 3 classes spécifiques pour 3 workers JPA et de deux méthodes de lecture dans *« FichierWorker »* pour lire deux fichiers de texte (localités et départements). Avec la technologie des *« generics »* apparus dans Java 5, on peut simplifier tout cela en *« typant »* génériquement une seule classe *« JpaDao »* pour créer les 3 workers JPA et idem avec *« FileDao »* en créant une seule classe typée avec des *« extracteurs »* spécialisées. On aimerait arriver à cela dans la couche *« métier »* :

**private JpaDaoItf<Personne, Integer> persJpa;**

**private FileDaoItf<Localite> ficLoc;**

**…**

**public DbWorker () throws MyDBException {  
 persJpa = new JpaDao<>(JPA\_PU, Personne.class);  
 locJpa = new JpaDao<>(JPA\_PU, Localite.class);  
 depJpa = new JpaDao<>(JPA\_PU, Departement.class);  
 ficLoc = new FileDao<>(new LocaliteExtracter("\t"));  
 ficDep = new FileDao<>(new DepartementExtracter(";"));**

**}**

La couche *« métier »* (DbWorker) permet de cacher complétement au *« contrôleur »* (MainCtrl) tous les accès à la base de données et aux fichiers *« texte »*, de la même manière que la couche JPA cache tous les accès à JDBC.

Cette couche *« métier »* permet aussi d’effectuer des opérations plus complexes (généralement avec une transaction gérée manuellement) que celles comprises dans la couche d’accès aux données.

### Consignes :

* Dans NetBeans, copier le projet Ex07 en Ex09.
* On vous met à disposition tous les tests unitaires disponibles pour que vous puissiez tester vous-même les implémentations de ces classes. Les copier dans votre projet NetBeans.

|  |  |
| --- | --- |
|  | * Implémenter les classes dans le même ordre (donc commencer par FileDao). * Tester une classe après l’autre * A\_FileDaoTest essaye de lire les fichiers dans *«****data****»* : * *« TousLesTests »* pemet de lancer tous les tests une fois toutes les implémentations effectuées |

Rappelons que les *« Generics »* permettent de s'abstraire du type réel des objets lors de la conception d'une classe ou d'une méthode tout en conservant un code sécurisé.

lls permettent de définir le type réel des objets lors de leur utilisation et permettent ainsi de s'affranchir des multiples conversions (*« cast »*) peu pratiques et dangereuses en cas de mauvaise utilisation. Ainsi, les *« Generics »* augmentent la sécurité en reportant à la compilation des erreurs qui survenaient à l'exécution.

* Créer un nouveau package *« app.workers.dao »* ensuite créer une nouvelle interface nommée *«****JpaDaoItf****»*. Cette classe utilise aussi les types paramétrés *« Generics »* amenés dans Java 5. *«****E****»* représente n’importe quel type *« d’entity-bean »* et *«****PK****»* le type de la PK utilisée :

**public interface JpaDaoItf<E, PK> {  
  
 // opérations CRUD  
 void creer( E e ) throws MyDBException;  
 E lire( PK pk ) throws MyDBException;  
 void modifier( E e ) throws MyDBException;  
 void effacer( PK pk ) throws MyDBException;  
  
 // opérations spéciales  
 long compter() throws MyDBException;  
 E rechercher(String prop, Object valeur) throws MyDBException;  
  
 // opérations sur des listes complètes de bean  
 List<E> lireListe() throws MyDBException;  
 int effacerListe() throws MyDBException;  
 int sauverListe( List<E> list ) throws MyDBException;**

**// opérations avec la BD**

**boolean estConnectee() ;  
 void deconnecter();  
}**

* Toujours dans *« app.workers.dao »*, créer une classe *«****JpaDao****»* qui implémente *«****JpaDaoItf****»* et copier-coller les méthodes spécifiques écrites auparavant pour une entity donnée (par exemple *« PersonneWrk »*). Adapter les méthodes pour les rendre *« génériques »*. Le constructeur doit avoir la signature suivante et connectera la base de données :

**public JpaDao( String pu, Class<E> cl ) throws MyDBException {**

**this.cl = cl;**

**try {**

**emf = Persistence.createEntityManagerFactory(pu);**

**em = emf.createEntityManager();**

**et = em.getTransaction();**

**} catch (Exception ex) {**

**throw new MyDBException(SystemLib.getFullMethodName(), ex.getMessage());**

**}  
 }**

La classe *« cl »* est nécessaire pour faire la recherche et la création des requêtes JPQL, car il n’est pas possible de connaître le type de E durant le fonctionnement du programme via une méthode système ou autre. Voici un exemple d’utilisation d’un named query dans la méthode *« rechercher »* à implémenter :

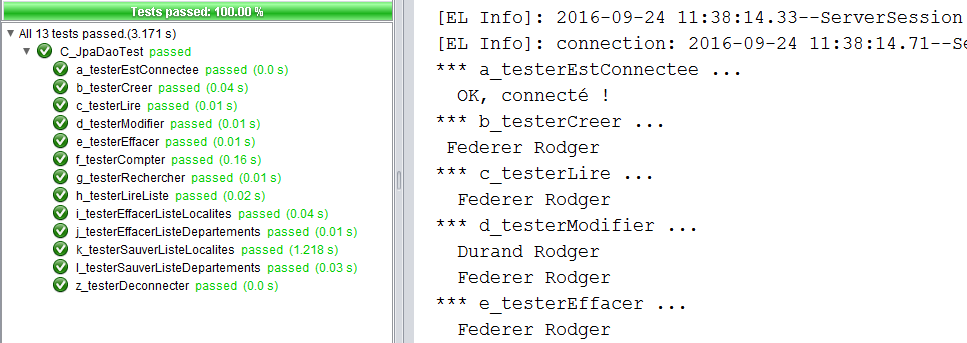
**String namedQuery = cl.getSimpleName() + ".findBy"  
 + prop.substring(0,1).toUpperCase()+prop.substring(1);  
 Query query = em.createNamedQuery(namedQuery);**

Il est aussi possible de créer les queries sans passer par les beans (c’est mieux) car s’ils sont regénérés on va perdre les modifications :

**Query query = em.createQuery("Select e from " +cl.getSimpleName() + " e");**

Notez la présence dans l’interface de 3 nouvelles méthodes *«****rechercher****»*, *«****effacerListe****»* et *«****sauverListe****»* qu’il faudra implémenter et tester les unes après les autres (méthodes de test fournies, voir plus loin). Ces 2 dernières utilisent pleinement la transaction car on exécute plusieurs modifications de la base de données avant de valider (commit). Cette manière de faire permet de diminuer les temps de modification d’un facteur 100, ce qui est bigrement intéressant. L’autre chose intéressante est qu’en cas d’erreur toutes les modifications sont annulées par le rollback et la base de données revient à son état initial.

* Tester au fur et à mesure l’écriture de vos méthodes JpaDao avec les méthodes de test fournies :



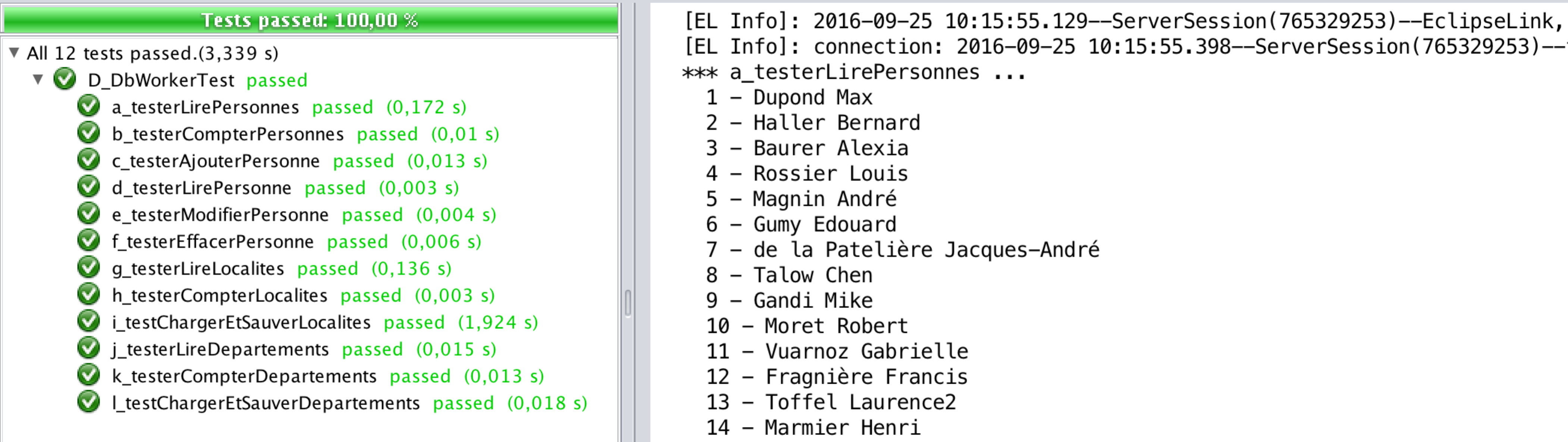
* Créer une interface *«****DbWorkerItf****»* avec les méthodes nécessaires à l’application :

**public interface DbWorkerItf {  
  
 List<Personne> lirePersonnes() throws MyDBException;  
 long compterPersonnes() throws MyDBException;  
 void ajouterPersonne( Personne p ) throws MyDBException;  
 Personne lirePersonne( Personne p ) throws MyDBException;  
 void modifierPersonne( Personne p ) throws MyDBException;  
 void effacerPersonne( Personne p ) throws MyDBException;  
  
 List<Localite> lireLocalites() throws MyDBException;  
 long compterLocalites() throws MyDBException;  
 int lireEtSauverLocalites( File fichier, String nomCharset ) throws Exception;  
  
 List<Departement> lireDepartements() throws MyDBException;  
 long compterDepartements()throws MyDBException;  
 int lireEtSauverDepartements(File fichier, String nomCharset) throws Exception;**

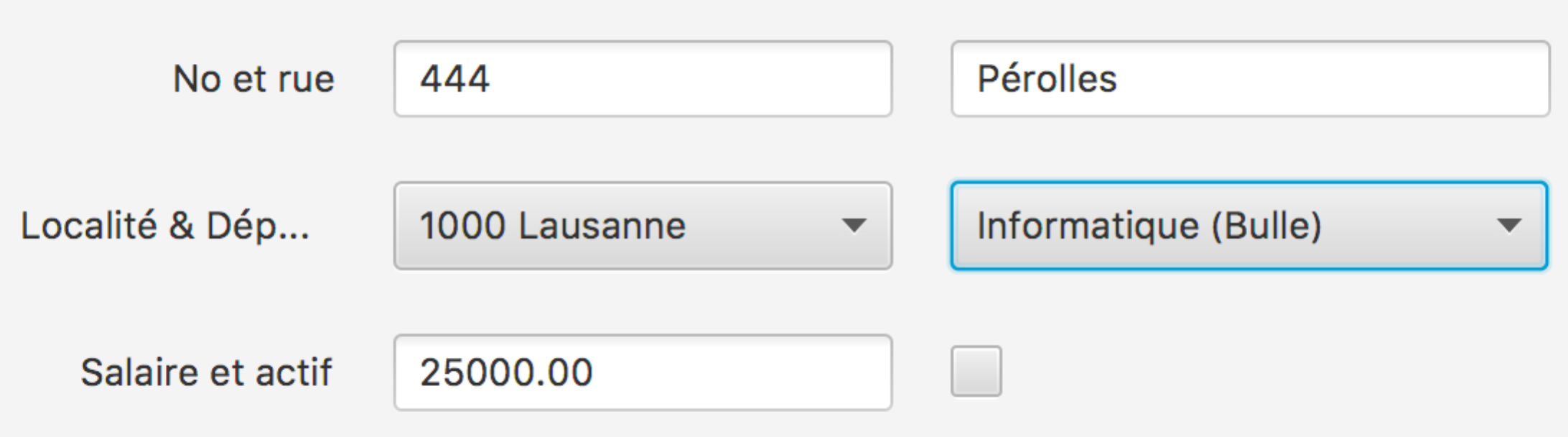
**boolean estConnecte();**

**void fermerBD();  
}**

* Créer la classe *«****DbWorker****»* dans *«****app.workers****»* et implémenter toutes les méthodes prévues dans l’interface *« DbWorkerItf ».* Notez que le constructeur de *« DbWorker »* commence exactement comme la méthode *« setUpClass »* de la classe de test *«B\_JpaDaoTest »*.
* Pour accéder aux entity-beans, il est nécessaire d’instancier un objet JpaDao par entity-bean, 3 fois dans notre cas. Cela implique plusieurs créations d’EntityManagerFactory. Si vous observez bien la console, il n’y a qu’une ouverture de base de données. JPA utilise un Singleton, comme à l’exercice précédent, pour éviter de multiples ouvertures de base de données qui sont très consommatrices de ressources. Par contre les EntityManagers sont différents, c’est une sorte de conteneur qui gère ses propres transactions. Cela permet le traitement concurrent sur un serveur Web. Ce qu’il ne faut pas oublier avant de quitter l’application, c’est de fermer les EntityManagers, 3x dans notre cas !
* Vérifier avec la classe de tests prévue :



* Relier la classe *«****MainCtrl****»* à *«****DbWorker****»*. IL NE DOIT PLUS RESTER DE REFERENCE DIRECTE A LA COUCHE JPA !!!
* Adapter l’Ihm (si pas déjà fait dans l’ex.7) pour permettre la gestion des localités et des départements dans des comboxBox :



Pour utiliser une comboBox avec Java FX8, voici quelques méthodes utiles :

**@FXML**

**private ComboBox<Localite> cmbLocalites;**

**…**

**cmbLocalites.getItems().addAll(dbWrk.lireLocalites());**

**…**

**p.setLocalite(cmbLocalites.getValue());**

**…  
 cmbLocalites.setValue(p.getLocalite());**

1. **Classe FileDao**

### Consignes :

* Créer un nouveau package nommé *«****app.workers.extracters****»*.
* L’interface *«****BeanExtracter****» de helper* vous permettra de normaliser l’extraction d’une ligne de texte (lue depuis un fichier) pour en faire un *« bean »*. Cette interface utilise déjà les *« generics »* de Java (E représente une classe-entité générique).

Exemple :

**public interface BeanExtracter<E> {  
  
 public E textToBean (String text);  
  
}**

* Dans *« app.workers.extracters »*, créer les deux classes DepartementExtracter et LocaliteExtracter qui implémentent *« BeanExtracter»* :



Le code doit juste créer un bean à partir d’une ligne de texte, ce qui donne pour les localités :

**public class LocaliteExtracter implements BeanExtracter<Localite> {  
 private String sep;  
  
 public LocaliteExtracter( String sep ) {  
 this.sep = sep;  
 }  
  
 @Override  
 public Localite textToBean (String text) {  
 String[] tab = text.split(sep);  
 Localite loc = new Localite();  
 if (tab.length > 6) {  
 loc.setNpa(Integer.parseInt(tab[2]));  
 loc.setPkLoc(loc.getNpa()\*100 + Integer.parseInt(tab[3]));  
 loc.setLocalite(tab[4]);  
 loc.setCanton(tab[6]);  
 }  
 return loc;  
 }}**

* Dans *« app.workers.dao »*, créer l’interface *«****FileDaoItf****»* comme suit :

**public interface FileDaoItf<E> {  
  
 public List<E> lireFichierTexte( File fichier, String nomCharSet )**

**throws MyFileException;  
  
}**

* Toujours dans *« app.workers.dao »*, créer la classe *«****FileDao****»* qui implémente les méthodes de l’interface *« FileDaoItf »* avec le code de lecture de fichier de l’exercice 7, mais en appelant la méthode *«****textToBean****»* d’un objet *« BeanExtracter»* passé en paramètre du constructeur :

**public FileDao( BeanExtracter<E> extracteur ) {  
 this.extracteur = extracteur;  
 }**

@Override  
 public List<E> lireFichierTexte( File fichier, String nomCharSet ) {

List<E> ar = new ArrayList<>();

**// prepare le bon charset (UTF8 ou autre) d'après le nom fourni**

**Charset cs = Charset.forName(nomCharSet);**

**// ouverture du fichier**

**try (BufferedReader reader = new BufferedReader(**

**new InputStreamReader(new FileInputStream(fichier), cs))) {**

**String line ;**

**while ((line = reader.readLine()) != null) {  
 E e = extracteur.textToBean(line);  
 ar.add(e);  
 }**

**} catch (…**

**...  
 }**

* Tester la lecture de fichiers texte avec la classe de test fournie :

