TP 1er programme Java

A. Saisie du code source de votre 1er programme Java :

1. Dans un éditeur basique (bloc notes ou vi) saisissez le code suivant dans un fichier nommé ***LanceurHelloWorld.java*** dans le répertoire de votre choix, en respectant bien la casse (majuscules, minuscules).

```
public class LanceurHelloWorld {

   public static void main(String[] args) {

       System.out.println( "Bonjour ! " );

   }

}
```

2. A partir d'une commande dos ou d'un terminal, se positionner dans le répertoire où se trouve le fichier source et compiler ce fichier LanceurHelloWorld.java via la commande :

   javac LanceurHelloWorld.java

3. Vérifier la présence du byte code associé dans le même répertoire:

   LanceurHelloWorld.class

4. Lancer le programme principal via la commande

   java LanceurHelloWorld

5. Vérifier l'affichage du résultat suivant :

   Bonjour !

B. Prise en compte des arguments de la ligne de commande.

1. Modifier le contenu du fichier source par le code suivant afin de gérer les arguments de la ligne de commande et manipuler les commentaires.

```
public class LanceurHelloWorld {

   /*
    *  La méthode main est statique.
    *  Ce qui signifie qu'elle est chargée en mémoire au démarrage
    *  de l'application et peut donc être utilisée avant la création de tout
    *  objet.
    */
   public static void main(String[] args) {

         //Affichage de l'argument saisi sur la ligne de commande
        System.out.println("Bonjour Monsieur "  + args[0]);

   }

}
```

2. Répéter les étapes 2. et 3. précédentes.

3. Lancer le programme principal via la commande :

java LanceurHelloWorld Smith

4. Vérifier l'affichage du résultat suivant :

Bonjour Monsieur Smith.

TP Packaging en ligne de commande avec Java Archive : JAR

A. Packaging de votre « application » en un fichier JAR:

1. A partir d'un fichier source, par exemple, nommé ***LanceurHelloWorld.java*** dans le répertoire de votre choix.

```
public class LanceurHelloWorld {

   /*
    *  La méthode main est statique.
    *  Ce qui signifie qu'elle est chargée en mémoire au démarrage
    *  de l'application et peut donc être utilisée avant la création de tout
    *  objet.
    */
   public static void main(String[] args) {

         //Affichage de l'argument saisi sur la ligne de commande
        System.out.println("Bonjour Monsieur "  + args[0]);

   }

}
```

Exécutez la commande suivante :

jar cvf hello.jar LanceurHelloWorld.class

Jar est une utilitaire du JDK pour packager une application. c, v et f sont des options de la commande : « c » pour create, « v » pour verbose et « f » pour file. N'hésitez pas à regarder les options en détails grâce à l'aide : ***jar -help***

Pour créer un fichier jar, il suffit donc de préciser le nom du fichier jar et le(s) nom(s) des fichiers compilés qui le composeront.

2. Vous pouvez lire le contenu d'un fichier JAR par la commande :

jar tvf hello.jar

Notez qu'un fichier MANIFEST.MF a été ajouté au jar dans un répertoire META-INF

Ce jar constitue une unité de livraison que vous pouvez livrer ou utiliser dans une autre application sous forme d'une librairie. C'est d'ailleurs comme cela que sont divisés et livrés les APIs des JRE.

B. Création d'un JAR exécutable

Le fichier JAR créé précédement n'est pas exécutable car nous n'avons pas précisé quel était le fichier qui doit être exécuté en premier (même s'il n'y a qu'un seul fichier, c'est obligatoire). C'est un fichier qui contient un « main » obligatoirement.

1. Dans le même répertoire, créer un fichier texte nommé ***manifestHelloWorld.txt***

2. Y écrire le code suivant pour indiquer que le fichier qui contient ce fameux « main » est notre fichier LanveurHelloWorld.java :

```
Manifect-Version: 1.0
Main-Class: LanceurHelloWorld
```

   Remarque :

   - Ce fichier doit se terminer par un retour à la ligne après '*LanceurHelloWorld'*, sinon cette ligne ne sera pas analysée. Attention également aux espaces après les « : » ainsi qu'à la casse (majuscules et minuscules).

3. Renommer ***manifestHelloWorld.txt*** en ***manifesteHelloWorld.mf***

4. Créer un jar comprenant ***LanceurHelloWorld.class*** et le fichier manifeste ***manifestHelloWorld.mf*** via la commande

   jar cmf manifestHelloWorld.mf hello.jar LanceurHelloWorld.class

5. Exécuter le jar ***hello.jar*** via la commande :

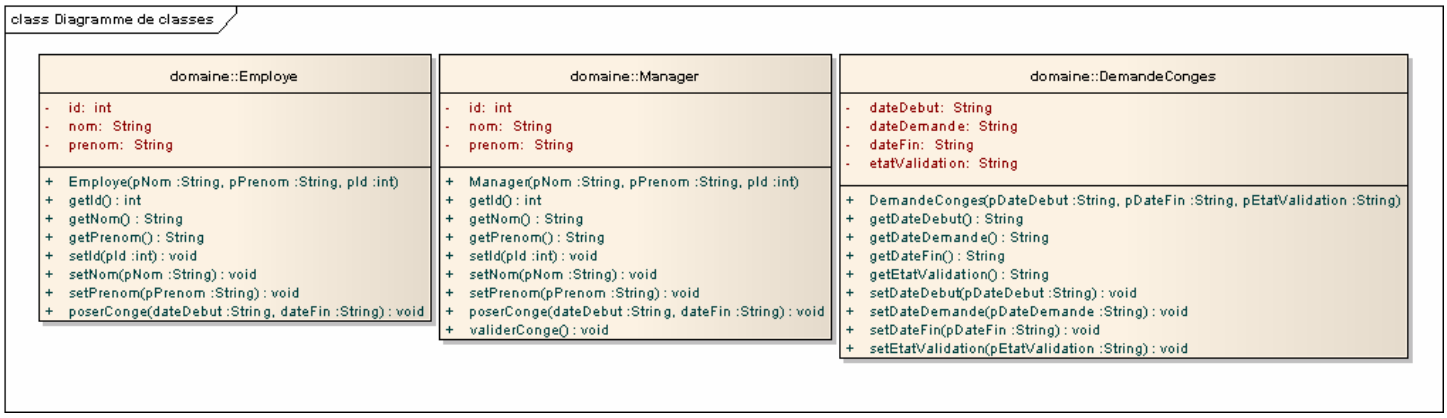
   java –jar hello.jar Smith

6. Vérifier l'affichage du résultat suivant :

   Bonjour Monsieur Smith.

TP « Implémentation d'un modèle UML »

L'objectif est d'implémenter le diagramme de classes UML suivant (c'est-à-dire, construire les classes avec leurs attributs, constructeurs et méthodes à partir de votre EDI).



Etapes :

1. Créer un package ***domaine***
2. Créer la classe ***Employe*** dans le package ***domaine***
3. Créer les attributs de la classe : *id*, *nom* et *prenom*
4. Créer le *constructeur* de la classe ***Employe***
5. Créer les accesseurs (méthodes qui commencent par « get » et « set »
6. Créer la méthode ***poserConge*** de la classe ***Employe*** en utilisant le code suivant pour faire un affichage des attributs de l'objet Employe et des paramètres passés en arguments de la méthode :

```
public void poserConge(String dateDebut, String dateFin) {
    System.out.println("Monsieur " + prenom + " " + nom + " pose un conge du "
                       + dateDebut + " au " + dateFin);
}
```

7. Pour les autres classes, ré-itérer l'opération mais ce coup-ci, utilisez les fonctionnalités de votre EDI pour générer automatiquement les constructeurs et accesseurs. Consultez l'aide (F1) ou si besoin, demandez au formateur. Pour les méthodes ***poseConge*** et ***validerConge***, contentez vous de faire un affichage comme dans l'exemple ci-dessus.
8. Créer un package ***lanceur***
9. Ecrire dans ce package la classe ***LanceurGestionConges*** (Classe comportant un Main) afin d'instancier un Manager et deux Employes.
10. Tester l'invocation des différentes méthodes sur les différentes instances d' Employe et de Manager.

TP « L'Héritage »

L'objectif est d'implémenter l'héritage et d'en comprendre l'utilité. Voici le diagramme UML correspondant :

Etapes :

1. La classe ***Manager*** hérite de la classe ***Employe***. Dans le code cela se traduit par le mot clé ***extends***:

```
public class Manager extends Employe {
    ...
}
```

2. La classe ***Manager*** n'a aucun attribut étant donné qu'elle hérite de la classe ***Employe***.

3. De même pour la méthode ***poserConge( ).***

4. Attention, afin de faciliter la maintenance du code, il faut que le constructeur de la classe ***Manager*** utilise celui de la classe ***Employe*** grâce au mot clé ***super*** :

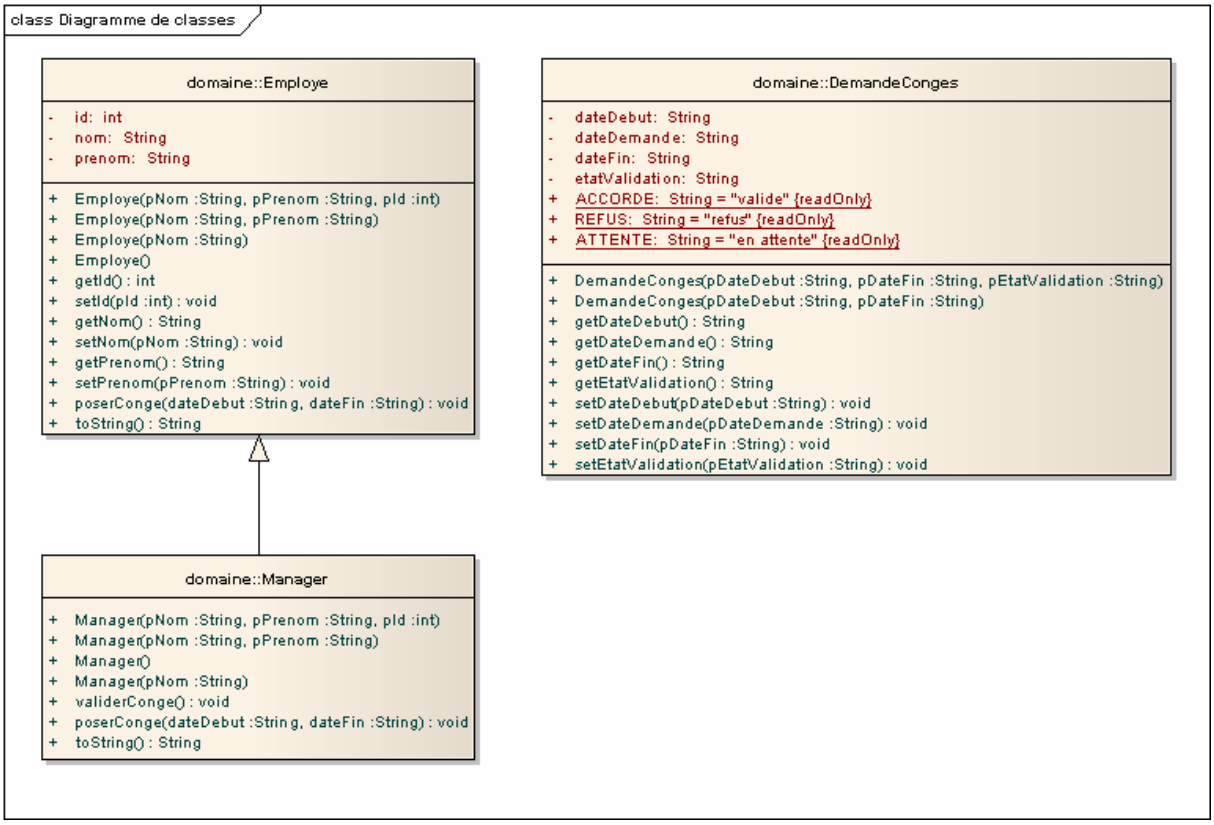
```
public Manager (String pNom, String pPrenom, int pId){
            super(pNom, pPrenom, pId);
}
```

Conclusion :

L'héritage a permis de factoriser des attributs communs : ***id, nom*** *et* ***prénom*** ainsi qu'une méthode commune : ***poserConge( )*** et les constructeurs.

TP « Redéfinition, Surcharge et Polymorphisme »

L'objectif est d'implémenter la redéfinition, la surcharge et le polymorphisme et d'en comprendre l'utilité. Voici le diagramme UML correspondant :



Etapes :

1. Ajouter à la classe ***DemandeConges***, trois attributs qui sont static (accessible à partir de la classe et nom d'un objet) et qui sont des constantes pour représenter les valeurs possibles des états d'une demande de congés :

```
public final static String REFUS = "refus";
public final static String ACCORDE = "valide";
public final static String ATTENTE = "en attente";
```

2. **La surcharge** d'une méthode ou d'un constructeur permet de définir plusieurs fois une même méthode/constructeur avec des arguments différents. Le compilateur choisi la méthode qui doit être appelée en fonction du nombre et du type des arguments. Ceci permet de simplifier l'interface des classes vis à vis des autres classes.

   Surcharger les constructeurs de *Employe*, *Manager* et *DemandeConges* via le mot clef java ***this***

   Exemple 1:

```
public Manager (String pNom, String pPrenom, int pId){
                super(pNom, pPrenom, pId);
}

public Manager (String pNom, String pPrenom){
                this(pNom, pPrenom, 0);
}
```

```
public Manager (String pNom){
        this(pNom, « Richard », 0);
}
public Manager (){
        this(« durant », « Richard », 0);
}
```

Exemple 2:

```
public DemandeConges(String pDateDebut, String pDateFin, String pEtatValidation) {
        dateDebut       =   pDateDebut;
        dateFin         =   pDateFin;
        etatValidation  =   pEetatValidation;
}

public DemandeConges(String pDateDebut, String pDateFin){
        this(pDateDebut, pDateFin, DemandeConges.ATTENTE);
}
```

Pourquoi utilise-t-on parfois ***this*** et d'autres fois ***super*** ?

3. **La redéfinition** d'une méthode héritée doit impérativement conserver la déclaration de la méthode parent (type et nombre de paramètres ainsi que la valeur de retour doivent être identique).

   Si la signature de la méthode change, ce n'est plus une redéfinition mais une surcharge. Cette nouvelle méthode n'est pas héritée : la classe mère ne possède pas de méthode possédant cette signature.

   La redéfinition de la méthode ***poserConges*** dans la classe ***Manager*** exprime le fait que le savoir faire métier est différent dans les deux classes :

   - La méthode ***poserConge()*** pour l'employé déclenchera une demande avec une attente de validation.
   - La méthode ***poserConge()*** pour le manager déclenchera une demande qui est automatiquement validée.
   - Pour l'instant, nous ne faisons la différence que par le biais d'un affichage différent dans les méthodes ***poserConge()*** d'Employe et Manager

   Exemple:

```
public void poserConge(String dateDebut, String dateFin) {
        System.out.println("Monsieur " + getPrenom() + " " + getNom() +
                           " pose un conge du " + dateDebut + " au " + dateFin);
        System.out.println("Etant manager, il valide ses propres conges");
}
```

4. On constate qu'une partie du code de ***poserConge()*** est identique dans les deux classes ***Employe*** et ***Manager***. Pour éviter la duplication de code, une bonne façon de faire est la suivante, toujours pour faciliter la maintenance :

```
public void poserConge(String dateDebut, String dateFin) {
        super.poserConge(dateDebut, dateFin) ;
        System.out.println("Etant manager, il valide ses propres conges");
}
```

5. Redéfinir la méthode ***toString()*** héritée de la super classe ***Object***, afin de tracer les caractéristiques d'un objet :

   Exemple:

```
public String toString() {
        return "Employe : " + id + ", " + prenom + ", " + nom ;
}
```

   Remarque:

   La méthode *toString()* est automatiquement invoquée lorsque l'on souhaite tracer un objet.

   Exemple:

```
Employe emp = new Employe("Smith" , " John", 2) ;
System.out.println(emp) ;
```

   Quand on passe une référence vers un objet à la méthode println(), celle-ci appelle automatiquement la méthode toString() sur la référence. L'exemple renverra donc la chaîne *:*

   Employe : 2, John, Smith

   Par défaut la méthode ***toString()*** de la classe ***Object*** retourne le nom de la classe d'instanciation de l'objet suivi de sa référence.

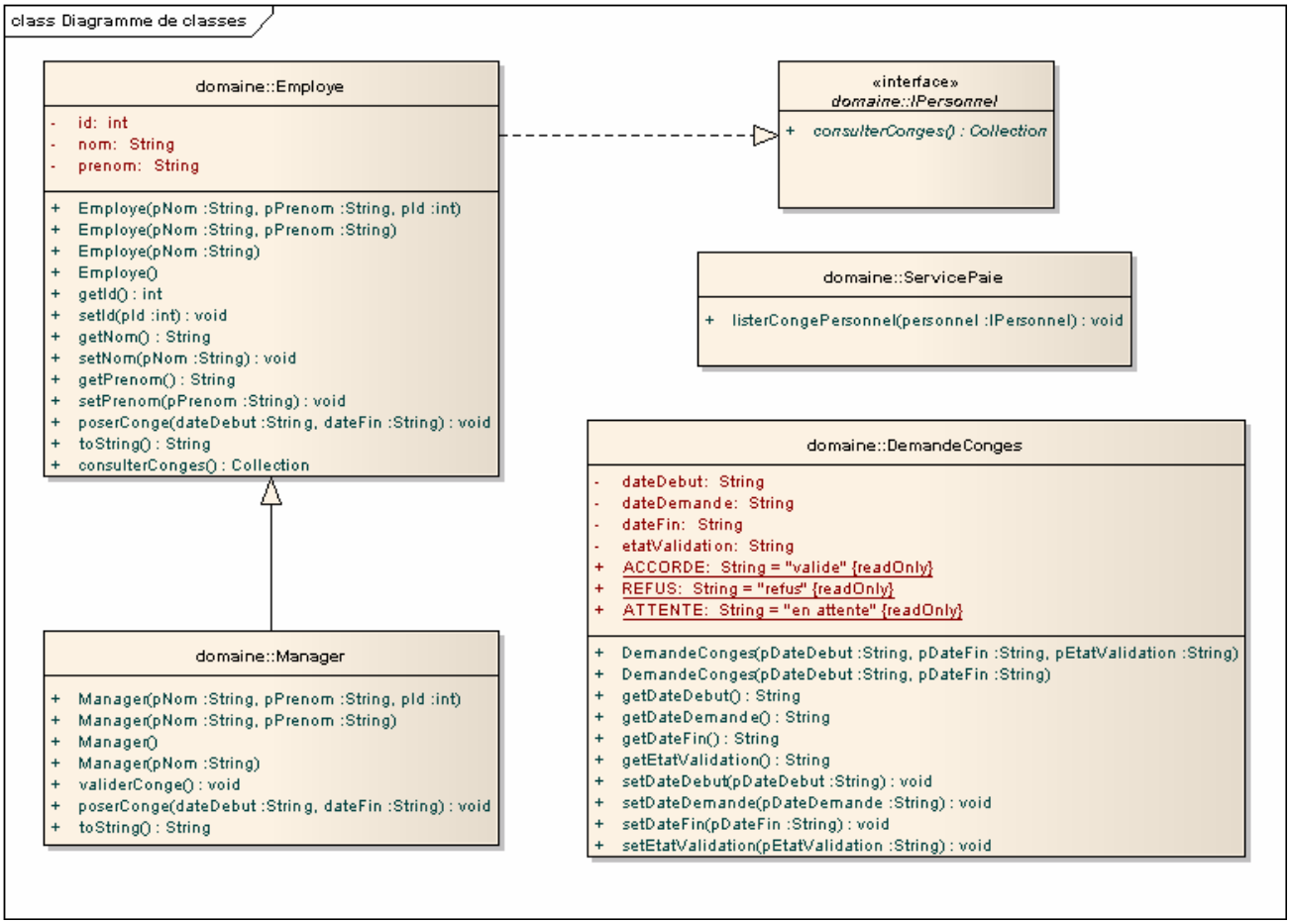
6. Tester à partir du lanceur la surcharge, redéfinition et polymorphisme par appels des différentes méthodes.

Conclusion :

La surcharge permet d'offrir plusieurs méthodes avec des paramètres différents tout en factorisant le code commun.
La redéfinition permet de redéfinir le comportement d'une méthode dans une classe fille (héritage).

TP « Utilisation des interfaces »

L'objectif est d'implémenter une interface afin d'en comprendre l'utilité. Voici le diagramme UML correspondant :



Objectif:

On considère un acteur tiers, la classe ***ServicePaie***, qui doit récupérer la liste de tout le personnel à la fin de chaque mois afin d'établir les feuilles de paie de chacun.

Pour cela, nous allons faire en sorte que les classes ***Employe*** et ***Manager*** (par héritage) implémentent une interface commune : ***IPersonnel***. Cette interface sera utilisée par la classe ***ServicePaie***. Le but étant de pouvoir à l'avenir modifier Employe, Manager ou même ajouter une nouvelle classe Directeur par exemple, sans impacter, sans modifier la classe ***ServicePaie***. On utilise ici également le ***polymorphisme***.

Etapes:

1. Créer l'interface ***IPersonnel*** dans le package ***domaine*** ayant la méthode ***consulterConges***().

```
public Collection consulterConges();
```

Cette méthode permet de consulter les congés posés par un Employé ou par un Manager.
L'idée est que cette interface puisse être utilisée par une classe tiers ***ServicePaie*** que vous allez implémenter.
La classe Collection permet de remplacer les tableaux notamment parce que sa taille est dynamique.

2. Faire en sorte que la classe ***Employe*** implémente l'interface ***IPersonnel*** à l'aide du mot clé « ***implements*** ».

```
public class Employe implements Ipersonnel {
    ...
}
```

Remarque :

La classe ***Manager,*** héritant de la classe ***Employe,*** est vue également de façon implicite comme une implémentation de l'interface ***IPersonnel.***

L'idée est donc qu'un objet ***Employe*** ou ***Manager*** puisse être manipulé par le biais de son interface ***IPersonnel*** (polymorphisme).

3. Redéfinir la méthode ***consulterConges()***

Exemple :

```
public Collection consulterConges() {
    System.out.println("Consultation des conges de " + prenom + " " + nom);
    return null;
}
```

4. Créer la classe ***ServicePaie*** dans le package ***domaine***.
Cette classe possède le savoir faire métier ***listerCongePersonnel***.

```
public void listerCongePersonnel(IPersonnel personnel){
    personnel.consulterConges();
}
```

Cette méthode demande l'affichage des congés d'un membre du personnel.

5. Modifier le lanceur ***LanceurGestionConges*** afin de :

- Créer une instance de ***ServicePaie***.
- Invoquer sur cette instance la méthode ***listerCongePersonnel*** en lui passant successivement une instance d'***Employe*** et de ***Manager***

Conclusion :

L'interface ici nous sert de contrat entre ***Employe***, ***Manager*** et ***ServicePaie***. Dans ServicePaie, la seule méthode que l'on peut utiliser sur une référence d'un Employe ou Manager, c'est la méthode ***consulterConges()*** . C'est d'ailleurs un plus en terme de sécurité.
Elle nous permet également de prévoir les futures évolutions, notamment l'ajout d'autres classes : ***DirecteurTechnique***, ***DirecteurCommercial*** sans pour autant modifier la classe ***ServicePaie***,

TP « Persistance à l'aide d'un fichier »

Objectif:

L'objectif est de sauvegarder vos données dans un fichier texte.

Etapes:

1. Implémener une méthode *exporter* qui sauvegarde dans un fichier **texte**, vos données applicatives.

Exemple :

Ajouter une méthode ***exporterConges()*** dans la classe ***ServicePaie*** dont la signature est :

public void exporterConges(IPersonnel emp)

Indications :

- Récupérer la liste des congés sous forme de ***Collection*** en invoquant la méthode ***consulterConges*** sur le paramètre de type ***IPersonnel***
- Utiliser la classe ***File*** pour obtenir une représentation java d'un fichier texte.
- Utiliser la classe ***FileWriter*** pour écrire dans ce fichier texte.

Tester la méthode ***exporterConges()*** via le ***LanceurGestionConges*** sur les instances suivantes :

```
ServicePaie paie = new ServicePaie();

Employe emp1 = new Employe("Watson", "John", 1);

paie.exporterConges(emp1);
```

Cette méthode doit écrire dans le fichier (*exportConges.txt)* la liste des congés de l'employé passé en paramètre à la méthode.

TP « Persistance à l'aide de la Sérialisation »

Objectif:

L'objectif est de sauvegarder vos données sous forme d'objets sérialisés dans un fichier.

Etapes:

1. Implémenter l'interface Serializable dans la classe des objets que vous souhaitez sérialiser.
2. Implémenter une méthode *exporter* qui sauvegarde dans un fichier, vos données applicatives mais sous forme d'objets.

Exemple :

Ajouter une méthode ***exporterConges()*** dans la classe ***ServicePaie*** dont la signature est :

public void exporterConges(IPersonnel emp)

Indications :

- Implémenter l'interface ***Serializable*** dans la classe ***DemandeConges***
- Récupérer la liste des congés sous forme de ***Collection*** en invoquant la méthode ***consulterConges*** sur le paramètre de type ***IPersonnel***.
- Utiliser la classe ***ObjectOutputStream*** et la méthode ***write(Object o)*** pour sérialiser des objets de type DemandeConges.

Tester la méthode ***exporterConges()*** via le ***LanceurGestionConges*** sur les instances suivantes :

```
ServicePaie paie = new ServicePaie();

Employe emp1 = new Employe("Watson", "John", 1);

paie.exporterConges(emp1);
```

Cette méthode doit écrire dans le fichier (***exportConges.txt)*** la liste des congés de l'employé passé en paramètre à la méthode.

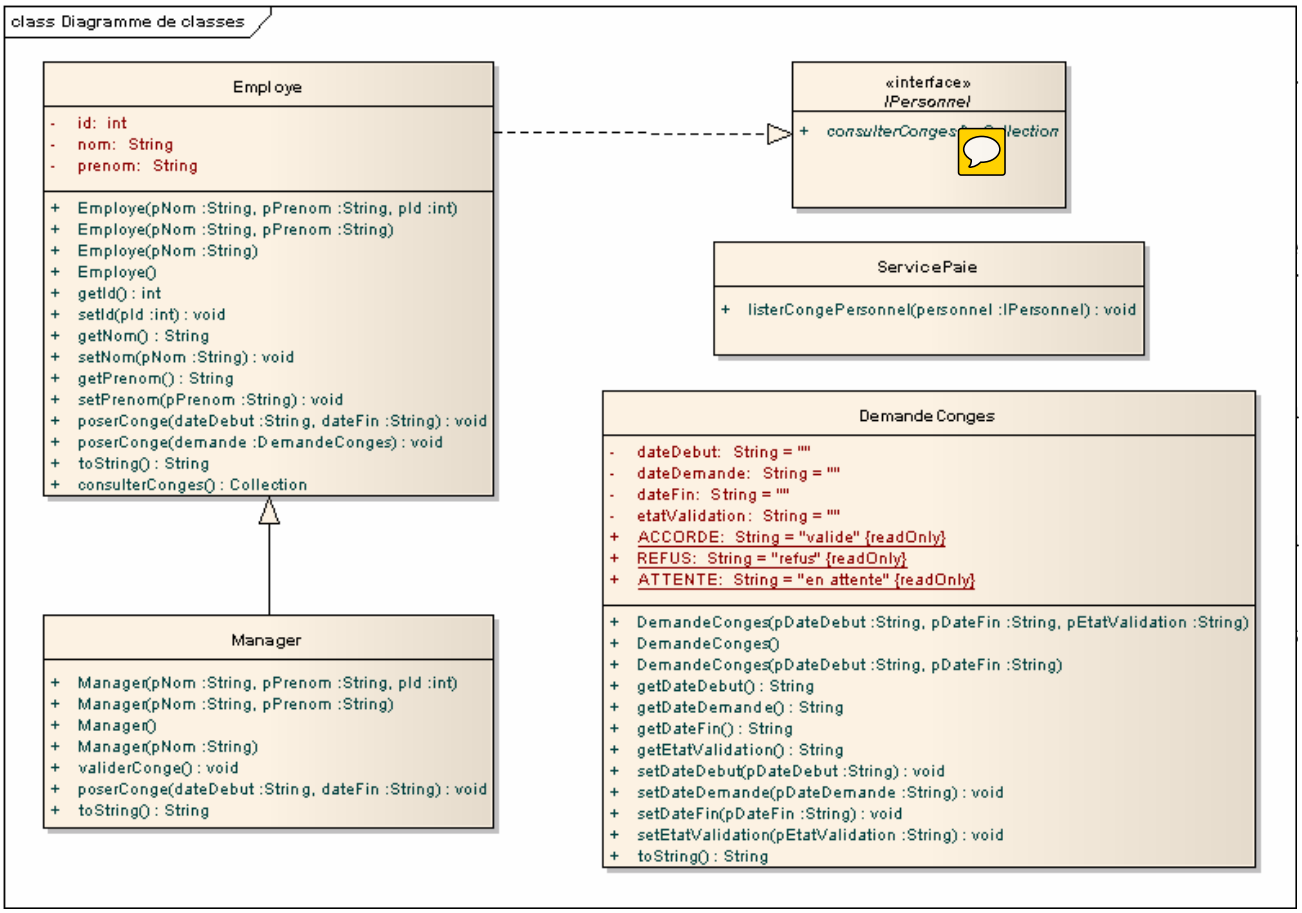
TP « Persistance a l'aide d'une BDD MySQL »

Objectif:

L'objectif est d'assurer la persistance des données d'une application à l'aide d'une BDD.

Etapes:

1. Implémener le diagramme de classe suivant pour préparer l'application à la persistance des données :



Remarque : On notera l'apparition d'une nouvelle méthode dans ***poserConge()*** qui prend en paramètre un objet de type ***DemandeConges***. C'est une surcharge de la méthode poserConges initiale.

Pour l'instant, ce n'est pas le sujet, nous allons d'abord implémenter la méthode ***consulterConges()*** qui dorrenavant ira chercher dans la BDD, la liste des congés des employés et retournera la liste des congés sous forme d'une collection. Nous verrons les collections plus tard, pour l'instant on se contentera de renvoyer null et d'afficher les demandes une à une dans la console.
Rappel, les classes dont vous aurez besoin se trouve dans l'API JDBC, c'est-à-dire ***java.sql***

1) Implémenter la méthode ***consulterConges()*** afin d'aller rechercher en base tous les congés posés par un membre du personnel identifié par son ID_employe.

   Pour cela, vous devez dans la méthode ***consulterConges()*** :

   a) Charger le driver Mysql en mémoire avec ***Class.forName(...)***

   b) Vous connecter à la BDD. Pour rappel avec ***DriverManager.getConnection(...)***:
      i) url : "jdbc:mysql://localhost/formation"
      ii) login : "root"
      iii) pawword : "" (une chaîne vide)

   c) Créer un objet Statement/PreparedStatement avec ***cn.createStatement();***

   d) Exécuter la requête avec ***st.executeQuery(sql)***
      La requête SQL à utiliser doit ressembler à :

      select * from conges where ID_employe=1 ;

   e) Parcourir le ResultSet obtenu avec une boucle sur ***rs.next()***

   f) Dans le parcours du ResultSet, se contenter pour le moment de mettre à jour et d'afficher une instance de ***DemandeConges*** avec les *dateDebut*, *dateFin* et *etat* récupérés de la base de données Pour les récupérer, utiliser les méthodes ***rs.getString(...)*** et ***rs.getInt(...)***

      Les champs de la table ont pour noms : ***dateDebut***, ***dateFin*** et ***etat,*** tous sont de type ***String***.

   g) Le tout doit être dans un bloc try/catch/finally. N'oubliez pas de libérer les resources avec la méthode ***close()*** qui doit être appellée sur le ***ResultSet***, le ***Statement*** et la ***Connection***.

2) Tester la méthode ***consulterConges( )*** via le ***LanceurGestionConges*** sur les instances suivantes :

```
Employe emp1 = new Employe("Watson", "John", 1);
Manager manager = new Manager("Homes", "Sherlock", 2);
```

   Vous devriez obtenir un résultat similaire à celui-ci :

```
Consultation des conges de John Watson
DemandeConges : 2005-03-04, 2005-03-08, refus
DemandeConges : 2005-06-06, 2005-06-17, valide
DemandeConges : 2005-08-08, 2005-08-19, en attente
Consultation des conges de Sherlock Homes
DemandeConges : 2005-09-10, 2005-09-25, en attente
DemandeConges : 2005-07-01, 2005-07-15, valide
```

3) **OPTIONNEL :** Implémenter la méthode ***poserConge()*** afin d'aller insérer en base un nouveau congé en attente si c'est un employé et validé si c'est un manager.

On ré-itère les mêmes étapes, la seule différence est que l'on exécute la requête non pas avec ***executeQuery(…)*** mais avec ***executeUpdate(…)***

La requête SQL, quant à elle ressemble à :

insert into conges (ID_employe, dateDebut, dateFin, etat)
values (1 , '02/01/2007' , '31/01/2007' , 'valide');

Remarques :

- L'implémentation de la méthode ***validerConge()*** du ***Manager*** se fera dans le TP sur les *Collections*

- Cette façon de coder n'est pas une bonne pratique, car nous melons à des objets métiers des appels techniques vers une solution de persistance (Base de données *mySQL* dans notre cas).
  Si nous souhaitons évoluer vers une base Oracle, il faudrait adapter le code partout ou cela est nécessaire.
  Une bonne façon de faire serait de créer une couche d'abstraction intermédiaire permettant de dissocier la couche métier de la couche technique.
  Cela pourrait être fait par le biais d'une Factory et de DAO (Data Access Object).

- Enfin, nous constatons que chaque méthode procède à l'ouverture d'une connexion à la base de données et à sa fermeture, opérations coûteuses.
  Une bonne pratique dans ce cas serait de gérer des pools de connexions.

4) **OPTIONNEL :** Tester la méthode ***poserConges()*** via le ***LanceurGestionConges*** sur les instances suivantes :

```
Employe emp1 = new Employe("Watson", "John", 1);
Manager manager = new Manager("Homes", "Sherlock", 2);

DemandeConges cg1 = new DemandeConges("2006-01-02", "2006-01-04");
DemandeConges cg2 = new DemandeConges("2006-09-09", "2006-09-14");

emp1.poserConge(cg1);
manager.poserConge(cg2);
```

Conclusion :

L'API JDBC est très facile à utiliser mais, ne permet pas de gérer facilement les *transactions*, la gestion de *pools de connexions* qui sont des problématiques importantes et que l'on rencontre rapidement lorsque le nombre d'utilisateurs d'une application accroît.

Heureusement, il existe des solutions (Design Pattern ***DAO***), d'autres API (***JTA***, ***JPA***) et d'autres frameworks (***Hibernate*** et ***TopLink***)

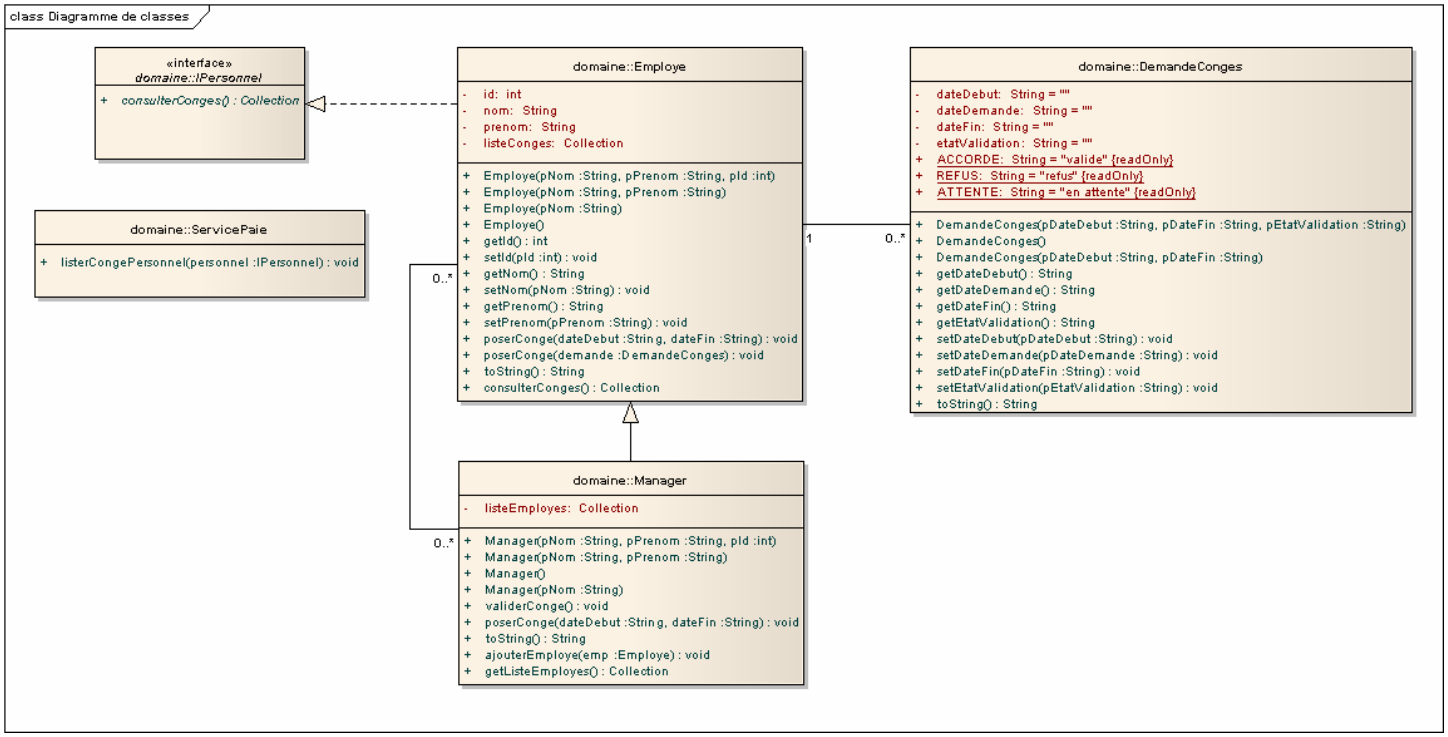
TP « Collections : les tableaux dynamiques»

Objectif:

L'objectif est d'utiliser les Collections, sorte de tableau dynamique qui en plus offre des fonctionnalités avancées (tri, conversions…).

Etapes:

1. Implémenter le diagramme de classe suivant pour finaliser l'application :



Remarques :

- Ajout d'un attribut **listeEmployes** de type ***Collection*** dans la classe ***Manager*** et de la méthode ***ajouterEmploye()***.
- Ajout de l'attribut ***listeConges*** de type ***Collection*** dans la classe ***Employe***.

2. Modifier la méthode ***consulterConges*** de la classe ***Employe*** afin de retourner une collection de ***DemandeConges***

Rappel : Jusqu'ici, la méthode se contente d'afficher les congés et de retourner la valeur ***null***.

```
Collection col = new ArrayList();
....
//Ajout de la demande dans la collection à retourner
col.add(demande);
```

3. Tester la récupération de la *Collection* retournée par méthode ***consulterConges()*** via le ***LanceurGestionConges***.

4. **Optionnel :** Implémenter la méthode ***ajouterEmploye(Employe emp)*** de la classe ***Manager***. Cette méthode ajoute l'employé ***emp*** passé en argument à la collection ***listeEmploye***.

```
public void ajouterEmploye (Employe e) {

        listeEmployes.add(e);

}
```

5. **Optionnel :** Modifier la méthode ***validerConge()*** de la classe ***Manager***.

Pour toutes les instances d'employés présentes dans la Collection ***listeEmployes***, procéder à la mise à jour de l'état des demandes de congés à «*valide* ».

Voici la requête de mise à jour :

UPDATE conges SET etat='valide' WHERE etat='attente' and ID_employe=1;

6. **Optionnel :** Tester la méthode ***validerConge()*** via le ***LanceurGestionConges*** sur les instances suivantes :

```
Employe emp1 = new Employe("Watson", "John", 1);
Manager manager = new Manager("Homes", "Sherlock", 2);
```

Penser à rattacher *emp1* à *manager* :

```
manager.ajouterEmploye(emp1);
```

Conclusion :

L'API des Collections propose un ensemble d'implémentations de listes : liste chaînées, liste non chaînées, dictionnaires qui sont indispensables.

TP « Traces applicatives avec Log4J»

Objectif:

Utiliser l'API Log4J pour gérer les traces de votre application.

Etapes:

1. Redémarrer Eclipse et utiliser le workspace suivant :

   \Ateliers\TP Log4J\workspaceLog4J

2. Ouvrir le premier projet et suivez les instructions du fichier « TODO.html ». Répondez aux questions.

3. Ouvrir le second projet et suivez les instructions du fichier « TODO.html ». Répondez aux questions.

4. Continuer jusqu'au dernier projet.

Conclusion :

L'API Log4J est tellement **utile** et **efficace** qu'elle est devenue presque un standard Java de facto.

TP « Configuration »

Objectif:

Configurer dynamiquement votre application (sans recompiler, ni redémarrer l'application).

Etapes:

1. En vous inspirant de la vidéo et de l'exemple fourni, modifier l'application pour mettre les paramètres dans un fichier de configuration (.properties) dans un répertoire « resources » de l'application.

Exemple : l'url, le nom du driver, le login et mot de passe, nécessaires pour se connecter à une BDD : ***resources/monfichier.properties***

```
bdd.url = jdbc:mysql://localhost/formation
bdd.driver=com.mysql.jdbc.Driver
bdd.login=root
bdd.password=
```

Noter qu'il n'est pas nécessaire de mettre des " " même pour les chaînes de caractères vides.

Conclusion :

La Classe Properties permet dynamiquement de lire un fichier de configuration.