TRAVAIL PRATIQUE 0

(exploration du logiciel JDeveloper 11g)

Ce petit travail a pour but premier de vous familiariser avec l'environnement de programmation JDeveloper 11g. Pour ce premier travail, vous devez d'abord compléter la classe `Employe`, servant à modéliser des objets dans le contexte du service de la paie d'une grande entreprise.

Débutez par copier dans votre espace personnel (J:) le dossier `ProjetTP0` situé dans le répertoire `Donnees:/B33`. Par la suite, ouvrez le projet (fichier `ProjetTP0.jpr`) à partir de JDeveloper (voir document d'aide de JDeveloper).

PARTIE A

1. Les données caractérisant les futurs objets `Employe` sont déjà inscrites, **vous n'avez rien à modifier et/ou à ajouter**. *Rappelez-vous que ces variables servent à représenter l'état d'un objet Employé*. Cependant, lorsque le besoin s'en fera sentir dans les exercices subséquents, vous devrez établir des constantes avec la méthode vue en classe à la suite de ces données.

2. Une méthode `Employe` est également présente : il s'agit d'une méthode constructeur. Nous reviendrons sur ce sujet la semaine prochaine. Toutes les méthodes à coder doivent être situées à la suite de cette méthode constructeur.

3. Concevez une méthode `salaireBrut`() permettant de retourner le salaire brut d'un employé, soit le nombre d'heures travaillées * le salaire horaire de cet employé.

4. Concevez une méthode `salaireNetAvantImpot` () retournant le salaire brut auquel on a enlevé les déductions suivantes :

déduction	montant
assurance – emploi	11,1 % du salaire brut
fonds de pension	1,36% du salaire brut
cotisation syndicale	montant fixe de 20 dollars

5. Concevez une méthode `impotFederal`() retournant le montant correspondant à l'impôt fédéral retenu. Il correspond à 19% du salaire net avant impôt.

6. Concevez une méthode `impotProvincial`() retournant le montant correspondant à l'impôt provincial retenu. Il correspond à 20% du salaire net avant impôt.

7. La méthode `salaireNetApresImpot`() correspond donc au salaire net avant impôt auquel on soustrait les montants correspondants à l'impôt fédéral et à l'impôt provincial.

8. La méthode `joursVacances` () sert à retourner le nombre de jours de vacances auquel un employé a droit. Ce nombre dépend de l'ancienneté de l'employé et de sa classe d'employé reflétée par son numéro d'employé. Le tableau suivant représente la situation :

classe d'employé	caractéristique	nombre de jours de vacances
A	le premier chiffre du numéro d'employé est un '1'	5 jours + 1 jour par année d'ancienneté
B	le premier chiffre du numéro d'employé est un '2'	10 jours + 1 jour par année d'ancienneté
C	le premier chiffre du numéro d'employé est un '3'	15 jours + 1 jour par année d'ancienneté
D	le premier chiffre du numéro d'employé est un '4'	20 jours + 1 jour par année d'ancienneté

9. Une dernière méthode, appelée `heuresSup`, permet d'ajouter un nombre d'heures supplémentaires passé en paramètre à la donnée `nbreHeuresSemaine`. Cette méthode ajuste donc le nombre d'heures travaillées et ne retourne rien.

PARTIE B

Une classe `TestEmploye` est également présente. Elle constitue une interface graphique permettant de saisir des informations sur un employé donné et d'afficher :

- son salaire net après impôt
- le nombre de jours de vacances auquel il a droit.

Vous pouvez ainsi vérifier si vos méthodes donnent le bon résultat.

PARTIE C : élaboration de classes de tests JUnit

La partie B vous a permis de vérifier, à un certain point, la fonctionnalité de votre classe `Employe`. Cependant, vous avez dû avoir recours à une interface graphique qui, si vous aviez eu à la faire, vous aurait pris pas mal de temps à réaliser. De plus, ce cours en est un de modélisation, où l'on veut le plus possible se détacher de la production de vues graphiques (pour l'instant).

Un moyen efficace d'effectuer des tests sur une ou des classes Java est d'employer le cadre JUnit. Ce cadre permet d'automatiser des classes de tests unitaires. Un test unitaire (*unit testing*) est un test qui s'oriente sur un module où il établit un environnement "artificiel" (création d'objets nécessaires au test, appels de méthodes. etc.).

Le fichier `testEmploye2` est une classe de tests JUnit. Elle a recours à deux librairies à importer.

La méthode `setUp` () sert à faire les initialisations nécessaires pour chacun des cas de tests qui suivront. Dans l'exemple, on crée un objet `Employe` appelé `unEmploye`. L'annotation `@Before` permet de garantir que la méthode setUp sera appelé avant chacune des méthodes de test; la méthode de test s'appuiera donc toujours sur un objet neuf

Au départ, il n'y a qu'un seul cas de test dans la classe `TestEmploye2` : `testSalaireBrut`. On y compare, à l'aide de la méthode `assertEquals` provenant des librairies JUnit, le **résultat donné par votre méthode** `salaireBrut` appliqué sur l'objet `unEmploye` avec **le résultat attendu**, soit 403.20. Le dernier paramètre constitue, dans le cas de valeurs décimales, l'écart qu'on permet pour que le résultat soit considéré égal à celui attendu.

Vous devez réaliser 4 autres cas de tests (méthodes). Les éléments à retenir pour construire un cas de tests JUnit :

Règles pour méthodes faisant partie du cadre JUnit SEULEMENT

1. la signature doit toujours être du même type : `public void …()` (pas de paramètres)
2. le nom de la méthode doit toujours débuter par l'annotation `@Test...`
3. vous comparez les résultats à l'aide de la méthode `assertEquals`. Si les résultats à comparer sont de type `double`, vous devez ajouter un troisième paramètre à la méthode `assertEquals`, ce dernier représentant l'écart admissible entre le résultat obtenu et espéré. Si ce troisième paramètre est 0, cela signifie que vous considérez le test valide uniquement si les résultats sont identiques

Me remettre dans LÉA (**date de remise : lundi 22 octobre**)

- L'ensemble de votre dossier ProjetTP0