

UE Programmation Orientée Objet

TP

Exercice 1 : Création dirigée de classe.

(Rappel : vous pouvez toujours ouvrir la fenêtre du terminal BLUEJ et activer l’option Record method calls pour visualiser les invocations JAVA)

Pour la création de code, notamment pour la syntaxe, vous vous inspirerez des exemples déjà vus lorsque le détail ne sera pas fourni.

Q 1 . Créez un nouveau projet avec BLUEJ (Projet → new Project) dans un répertoire tp2. Appelez le **Premier** par exemple.

Q 2 . Vous disposez maintenant d’une zone de projet ne contenant que l’icône du fichier `readme.txt`. Éditez ce fichier et indiquez-y les informations propres au projet (titre : **Premier**, votre nom, la date par exemple)

Q 3 . Maintenant créez une nouvelle classe que vous appellerez **Stock** (bouton New Class ou par le menu Edit), rappelons que par convention, les noms de classe commencent par une majuscule.

Q 4 . Ouvrez l’éditeur de code pour cette classe (rappel : double click sur l’icône de classe).

L’éditeur contient un squelette de déclaration de la classe.

Supprimez tout ce qui se trouve dans le corps de la classe (après `public class Stock` et entre les `{` et `}`).

Il vous faut maintenant effectivement définir la classe c’est-à-dire déterminer les fonctionnalités, c-à-d les *méthodes*, que l’on veut que la classe propose (et qui seront donc disponibles pour les instances de cette classe) et les *attributs* qui permettent la mise en œuvre de ces méthodes et la caractérisation de chacune des instances.

Dans cet exercice, la classe que nous allons concevoir doit permettre la modélisation de stocks (de marchandises par exemple) et leur gestion. On souhaite pouvoir consulter la quantité en stock et ajouter des éléments ou en retirer.

Cette dernière phrase fait apparaître les besoins fonctionnels de la classe : consultation, ajout et suppression. Ces opérations nécessitent une “mémoire” sous la forme d’un nombre, elle se traduit ici par un attribut numérique entier représentant la quantité du stock.

Q 5 . Créez l’attribut de type entier **quantite**, comme tous les attributs celui-ci doit se voir attribuer le modificateur `private` :

```
private int quantite;
```

Q 6 . Nous allons créer la méthode qui permet de consulter le stock, nous l’appellerons `getQuantite`, elle renvoie évidemment une valeur de type `int` et cette valeur doit tout aussi évidemment correspondre à la valeur de l’attribut **quantite** pour l’instance (objet) qui “exécute” cette méthode, on a donc :

```
public int getQuantite () {
    return this.quantite;
}
```

Q 7 . Pour pouvoir créer des objets de cette classe, il faut définir le constructeur. Rappelons qu’en JAVA il a le même nom que la classe. Le constructeur est généralement utilisé également pour initialiser la valeur des attributs, ici nous supposons qu’initialement le stock est vide, d’où :

```
public Stock () {
    this.quantite = 0;
}
```

Q 8 . Sauvegardez votre code, compilez le puis créez une instance pour le tester.

Q 9 . En vous inspirant des exemples de code déjà rencontrés lors du TP1, ajoutez des commentaires pour la méthode `getQuantite` et le constructeur. Ces commentaires sont encadrés par `/**` et `*/` et contiennent des marqueurs `@param` (pour la documentation sur un paramètre), `@return` (pour la documentation sur la valeur de retour quand il y en a une).

Rappel : vous pouvez visualiser vos commentaires en choisissant **Documentation** au lieu de **Code Source** dans la fenêtre où se trouve le code.

Q 10 . Il est possible que vous commettiez des erreurs dans votre code, dans ce cas vous serez prévenu à la compilation. Nous allons le vérifier : supprimez le “ ; ” à la fin de la ligne de code de la méthode `getQuantite`.

Vous avez peut être remarqué que dès que vous avez modifié le code source de la classe, l’instance existante a été détruite. Celle-ci risque en effet d’être “corrompue” par les modifications.

Sauvez et compilez : des erreurs sont mentionnées en bas de la fenêtre d’édition (message “ ; expected”) et la ligne (supposée) où se situe l’erreur est surlignée dans le code.

Corrigez l’erreur et recompilez.

Q 11 . Retournez dans le code et en vous inspirant de ce qui a été vu, ajoutez à la classe la méthode `ajoute` qui prend en paramètre un entier n et permet d’augmenter le stock actuel de n éléments. Cette méthode ne renvoie pas de résultat, le type de retour dans ce cas est `void`, son action est la modification de l’attribut **quantite**.

Ecrivez la javadoc, codez, compilez, testez.

Q 12 . On s’intéresse maintenant à la méthode `retire` qui permet de diminuer le stock en fonction de la valeur de son paramètre entier représentant la quantité à retirer du stock.

Il peut arriver que le stock existant courant soit inférieur à la quantité que l’on souhaite retirer. Dans ce cas évidemment il ne doit évidemment pas être possible de retirer plus que l’existant, la quantité du stock ne pouvant être négative. Cette méthode aura pour résultat un entier qui sera la quantité effectivement retirée du stock (et qui ne correspond pas forcément à la valeur du paramètre)¹.

Ne prévoyez pas d’affichage de message dans la méthode ! La valeur retournée par la méthode suffit comme information, si besoin on pourrait afficher le résultat retourné par la méthode, mais ce n’est pas à la méthode de le faire.

Ecrivez la javadoc, codez, compilez, testez.

Q 13 . Réaliser une méthode `toString` qui renvoie une chaîne de caractères telle que “la quantité en stock est de n ” où n représente la quantité du stock.

Q 14 . Ajoutez un second constructeur qui permet de créer un stock dont la valeur initiale n’est pas nulle. Cette valeur sera passée en paramètre du constructeur.

Exercice 2 : Autre classe, moins dirigée

On va définir une classe pour modéliser des marchandises. Chaque marchandise est définie par un nom et une valeur hors taxe en entier d’Euros.

Q 1 . Créez un nouveau projet, **Marchandise** par exemple.

Q 2 . Définissez pour cette classe, deux attributs, sans valeur initiale : **value** un `float` et **name** une instance de la classe `String`.

Q 3 . Définissez deux constructeurs :

- l’un n’a qu’un paramètre, il permet l’initialisation de l’attribut **name** et positionne l’attribut **value** à 0 par défaut. Quel est le type du paramètre ?
- l’autre prend comme paramètre les valeurs d’initialisation des deux attributs **name** et **value**. Quels sont donc les types des paramètres ?

Q 4 . Avez-vous écrit la JAVADOC pour les deux attributs et les constructeurs ? Si non faites le maintenant.

Q 5 . Ecrivez (et commentez) les méthodes d’accès (consultation) et de modification pour l’attribut **value**.

Q 6 . Idem pour uniquement l’accessor de **name**.

Q 7 . Codez une méthode `toString`, sans paramètre et qui retourne (pas affiche !) une chaîne de caractères dont la valeur est : “la marchandise **name** vaut **value**”. L’opérateur `+` permet la concaténation des chaînes de caractères avec conversion automatique des valeurs de types primitifs en valeurs de chaînes.

Q 8 . Codez (commentez !) une méthode qui retourne (**pas affiche !**) la valeur TTC de la marchandise, on prendra une TVA à 19.6%².

Q 9 . Testez.

Exercice 3 : Des ampoules... Une ampoule électrique est un objet de la vie quotidienne. Il est possible de l’allumer, de l’éteindre, etc. Elle est définie par des propriétés (puissance en watt, une couleur, allumée ou éteinte).

Q 1 . Créez un nouveau projet **Ampoule**.

Q 2 . Définissez dans BLUEJ, une classe **Ampoule**.

Q 3 . Définissez et écrivez les signatures (et uniquement les signatures) méthodes pour cette classe.

¹Donc si la quantité en stock est 7 et que l’on veut retirer 3, la méthode renvoie la valeur 3 et la quantité vaut ensuite 4 ; par contre si la quantité en stock est 7 et que l’on veut retirer 9, la méthode renvoie la valeur 7 et la quantité vaut ensuite 0.

²Pour la compatibilité des types vous utiliserez la notation `19.6f` où le `f` force le type de la constante à `float` et non à `double` comme c’est le cas par défaut

- Q 4 . Rédigez la JAVADOC de ces méthodes et visualisez là.
- Q 5 . Définissez dans le source les attributs nécessaires pour ces méthodes ainsi que le (ou les) constructeur(s).
- Q 6 . Codez le corps des méthodes de la classe.
- Q 7 . Construisez des instances et manipulez les pour faire des tests.

Exercice 4 : ... et des interrupteurs

On s'intéressera ici à la modélisation d'interrupteurs électriques qui permettent l'allumage et l'extinction d'une (et une seule) ampoule donnée. L'ampoule contrôlée par l'interrupteur est définie à la construction de l'objet interrupteur. On peut appuyer sur un interrupteur chaque appui fait passer l'ampoule de éteinte à allumée ou inversement.

- Q 1 . Créez une classe `Interrupteur` dans le projet `Ampoule` à l'aide de BLUEJ.
- Q 2 . Définissez l'interface publique de cette classe (c'est-à-dire les méthodes publiques de cette classe).
- Q 3 . Ecrivez la JAVADOC.
- Q 4 . Définissez les attributs (l'état) des instances de cette classe et son constructeur.
- Q 5 . Codez les méthodes.
- Q 6 . Testez.
- Q 7 . Créez une classe `InterrupteurMulti` qui permet de contrôler plusieurs ampoules à la fois. Le nombre maximum d'ampoules contrôlées est défini à la création de l'interrupteur et initialement il n'y aucune ampoule.

On utilisera un tableau pour gérer les ampoules contrôlées et on disposera d'une méthode :

```
/** fixe la (i-1)ème ampoule contrôlée par cet interrupteur. Il ne
 * se passe rien si i ne correspond pas à un indice valide
 * @param i l'indice de l'ampoule contrôlée (première ampoule à 0)
 * @param lAmpoule l'ampoule contrôlée
 */
public void ajouteAmpoule(int i, Ampoule lAmpoule)
```

La méthode qui gère l'appui sur l'interrupteur allumera ou éteindra chacune des ampoules contrôlées (rappel : une référence non initialisée vaut `null`, on peut le tester avec `==`).

On ajoute une méthode `afficheEtatAmpoules` qui pour chacune des ampoules contrôlées affiche (effet de bord) un message indiquant si cette ampoule est éteinte ou allumée.

Exercice 5 : Complexe

Ecrivez la classe `Complexe` vu en TD. N'oubliez pas la documentation, ni les tests!