[](http://lavande.cpe.fr/intranet/images/logo_CPE/logo_CPE_2009_moyenne_def.jpg)

Année universitaire 2015/2016

NOM :…………………………………… PRENOM :……………………………

Consignes relatives au déroulement de l’épreuve

Date : 19 janvier 2016

Contrôle de : Module 4ETI de Programmation Objet / Java, 1ère partie (1ère session)

Durée : 1h 15 minutes.

Professeur responsable : Martine BREDA

Documents écrits (supports de cours, fascicules d'exercices, corrections de TP...) sont autorisés 🗷   
Les **livres ne sont pas** autorisés.

LES CALCULATRICES, TELEPHONES PORTABLES ET AUTRES APPAREILS DE STOCKAGE DE DONNEES NUMERIQUES SONT **INTERDITS**.

Les téléphones portables doivent être éteints pendant toute la durée de l’épreuve et rangés dans les cartables.

Les oreilles des candidats doivent être dégagées.

Rappels importants sur la discipline lors des examens

La présence à tous les examens est strictement obligatoire ; tout élève présent à une épreuve doit rendre une copie, même blanche, portant son nom, son prénom et la nature de l’épreuve.

Une absence non justifiée à un examen invalide automatiquement le module concerné.

Toute suspicion sur la régularité et le caractère équitable d’une épreuve est signalée à la direction des études qui pourra décider l’annulation de l’épreuve; tous les élèves concernés par l’épreuve sont alors convoqués à une épreuve de remplacement à une date fixée par le responsable d’année.

Toute fraude ou tentative de fraude est portée à la connaissance de la direction des études qui pourra réunir le Conseil de Discipline. Les sanctions prises peuvent aller jusqu’à l’exclusion définitive du (des) élève(s) mis en cause.

Consignes générales :

Répondre sur ce document.

Lire soigneusement l’énoncé et les annexes avant de commencer l’exercice.

Pour les instructions Java, la syntaxe doit rester précise et compréhensible même si elle n'est pas tout à fait exacte.

Pour les questions qui attendent des réponses en français, ne récitez pas le cours de manière théorique mais soyez **synthétique** et **précis**.

Problème :

*Inspiré par : https://class.coursera.org/intropoojava-001*

Une montre est un produit composé d'un mécanisme qui peut être digital, analogique ou mixte et de différents accessoires (un boitier, un bracelet, un fermoir, une vitre).

Les mécanismes et les accessoires sont aussi des produits : on peut les acheter séparément.

Tous les produits ont un prix dont le calcul peut varier à partir d'une valeur de base :

* le prix d'un bracelet est sa valeur de base.
* le prix d'une montre est la somme des prix de de ses composants.

Tous les produits doivent pouvoir être affichés selon leur composition.

Exemples d'affichage :

* un bracelet :

bracelet cuir 54.0

* une montre :

Une montre composée de :

\* Mécanisme double (affichage : sur l'écran : 9:15, date 20140401, sous les aiguilles : réveil 7:00), prix de l'ensemble mécanisme double : 468.0

\* bracelet cuir 54.0

\* fermoir acier 12.5

\* boitier acier 36.6

\* vitre quartz 44.8

==> Prix total : 615.9

Le schéma ci-dessous représente les différentes classes et les liens d'héritage entre elles. (Les liens de composition n'y figurent pas).



**Question 1 :** Parmi ces classes dites lesquelles vous proposez de déclarer abstraites? Justifier brièvement vos choix.

|  |
| --- |
|  |

La classe Produit est définie comme suit en première approche.

**Question 2 :** Ecrire le code des accesseurs/mutateurs qui vous paraissent pertinents.

|  |  |
| --- | --- |
| abstract class Produit **implements** Cloneable {  // un produit a un prix de base et un nom  private final double prix;  private final String nom;  **public** Produit(**double** uneValeur, String unNom) {  **this**.prix = uneValeur;  **this**.nom = unNom;  }  **public** Produit() {  **this**.prix = 0.0;  **this**.nom ="";  }  **public** Object clone() {  **try** {  **return** **super**.clone();  } **catch** (CloneNotSupportedException e) {  e.printStackTrace();  }  **return** **null**;  } |  |

Justifiez vos choix.

|  |
| --- |
|  |

Les classes qui décrivent les différents mécanismes sont les suivantes :

//======================================================================

**abstract class Mecanisme extends Produit {**

//...

}

//======================================================================

**class** MecanismeAnalogique **extends** Mecanisme {

// ...

}

//======================================================================

**interface** ReveilDigital{

String toStringReveil();

}

//======================================================================

**class** MecanismeDigital **extends** Mecanisme **implements** ReveilDigital {

// ...

}

//======================================================================

**class** MecanismeDouble **extends** MecanismeAnalogique **implements** ReveilDigital {

// ...

}

**Nous ne les étudierons pas dans ce devoir.** **Par contre, nous allons compléter maintenant les classes Montre, Accessoire et classes dérivées.**

Dans une première approche, ces classes sont définies comme ci-dessous.**.**

//======================================================================

**class** Montre **extends** Produit {

**private** Mecanisme coeur;

**private** List<Accessoire> accessoires;

**public** Montre(Mecanisme depart)

{

coeur = depart.clone();

accessoires = **new** ArrayList<Accessoire>();

}

}

//======================================================================

**abstract** **class** Accessoire **extends** Produit {

**public** Accessoire(**double** valeurDeBase, String unNom){

**super**(valeurDeBase, unNom);

}

}

//------------------------------------------------------------

**class** Bracelet **extends** Accessoire {

**public** Bracelet(**double** valeurDeBase, String unNom) {

**super**(valeurDeBase, "bracelet " + unNom );  
}

}

//------------------------------------------------------------

**class** Fermoir **extends** Accessoire {

**public** Fermoir( **double** valeurDeBase, String unNom) {

**super**(valeurDeBase, "fermoir " + unNom);  
}

}

//------------------------------------------------------------

**class** Boitier **extends** Accessoire {

**public** Boitier( **double** valeurDeBase, String unNom) {

**super**(valeurDeBase, "boitier " + unNom);

}

}

//------------------------------------------------------------

**class** Vitre **extends** Accessoire {

**public** Vitre( **double** valeurDeBase, String unNom) {

**super**(valeurDeBase,"vitre " + unNom);

}

}

**Question 3 :**

L'instruction Produit p = new Produit(); est-elle correcte?

Justifier votre réponse.

|  |
| --- |
|  |

**Question 4 :**

Quelle méthode donne le prix d'un produit, quel que soit le type réel de ce produit?

|  |
| --- |
|  |

Dans l'état actuel, est-ce que cela fonctionne pour TOUS les types de produit? Expliquer brièvement.

|  |
| --- |
|  |

Sinon, quelle classe devez-vous compléter?

|  |
| --- |
|  |

Ecrire le code de la fonction que vous devez redéfinir dans la classe concernée.

|  |
| --- |
|  |

Quel mécanisme avez-vous mis en œuvre?

|  |
| --- |
|  |

**Question 5 :**

Pour que tous les produits soient affichables mais chacun avec ses spécificités :

* Dans quelles classes faut-il redéfinir la méthode toString() de la classe Objet pour avoir un affichage polymorphe?

|  |
| --- |
|  |

* Quelle précaution faut-il prendre pour afficher le **prix réel** d'un produit?

|  |
| --- |
|  |

* Ecrire le code de la fonction toString() pour la classe Produit.

|  |
| --- |
|  |

**Question 6 :**

On veut pouvoir faire une copie d'une montre.

Quelle(s) classe(s) faut-il compléter?

|  |
| --- |
|  |

Ecrire le code la méthode que vous proposez d'ajouter.

|  |
| --- |
|  |

**Question 7 :**

On veut pouvoir ajouter un accessoire sur une montre.

|  |
| --- |
| Par exemple : Montre m = **new** Montre(**new** MecanismeDouble(468.00, "9:15", 20140401, "7:00"));  Accessoire braceletCuir =**new** Bracelet(54.0, "cuir");  m.ajouter(braceletCuir);  m.ajouter(**new** Fermoir(12.5, "acier"));  m.ajouter(**new** Boitier(36.60, "acier"));  m.ajouter(**new** Vitre(44.80, "quartz")); |

Ecrire la méthode ajouter() de la classe Montre.

|  |
| --- |
|  |

**Question 8 :**

On veut aussi pouvoir changer un accessoire, c'est-à-dire en retirer un et le remplacer par un autre.

|  |
| --- |
| Par exemple pour changer le bracelet de la montre précédente :  Accessoire braceletMode = **new** Bracelet(20.0, "mode");  m.changer(braceletCuir, braceletMode); |

Pour identifier l'accessoire qu'on veut retirer de la liste d'accessoires, deux méthodes doivent être ajoutées à la classe Produit? Lesquelles? Pourquoi?

|  |
| --- |
|  |

Ecrire les méthodes retirer( ) et changer( ) de la classe Montre qui utilisent les méthodes ci-dessus.

|  |
| --- |
|  |

ANNEXES : La classe *Object*

|  |  |
| --- | --- |
| public boolean equals(Object obj) | Retourne *true* si un objet est égal à un autre. |
| public int hashCode() | Retourne une valeur de *hash-code* pour l’objet courant. |
| public String toString() | Retourne une représentation en chaîne de l’objet : la définition pour Object est getClass().getName() + '@' + Integer.toHexString(hashCode()). |
| public Class getClass() | Retourne la classe de l’objet. |
| protected  Object clone() | Création d’une copie de l’objet.   * Si l’objet n’implémente pas l’interface Cloneable, alors une exception  CloneNotSupportedException est levée. * Sinon la méthode crée une nouvelle instance de la classe et initialise tous ses attributs avec les valeurs des attributs de l’objet : si l’attribut est une référence c’est la référence qui est copiée : il n’y a pas copie en profondeur. |

L’interface List

|  |  |
| --- | --- |
| void add(int ind, E elt) | Insère elt à l’indice ind (opération optionnelle) |
| boolean addAll(int ind,  Collection<? extends E> c2) | Insère tous les éléments de la collection c2 à partir de l’indice ind.(opération optionnelle) |
| get(int ind) | Retourne l’élément à la position ind. Peut lever une exception IndexOutOfBoundsException si ind n'est pas correct. |
| int indexOf(Object o) | Retourne l’indice de la première occurrence de o, ou -1 si la list ne contient pas o. |
| int lastIndexOf(Object o) | Retourne l’indice de la dernière occurrence de o, ou -1 si la list ne contient pas o. |
| ListIterator<E> listIterator() | Retourne un ListIterator sur la liste |
| ListIterator<E> listIterator(int ind) | Retourne un ListIterator sur la liste, démarrant à l’indice ind. Peut lever une exception IndexOutOfBoundsException si ind n'est pas correct. |
| E remove(int ind) | Enlève l'élément à l'indice ind, et retourne cet élément. Peut lever une exception IndexOutOfBoundsException si ind n'est pas correct. |

L’interface Iterator

|  |  |
| --- | --- |
| boolean hasNext() | Retourne *true* si l’itérateur n’est pas arrivé en fin de l’ensemble et *false* sinon. |
| E next() | Permet d’avancer l’itérateur, et retourne l’élément qui était obtenu avant ce déplacement.  Lève une exception *NoSuchElementException* si l'itérateur n'a pas d'objet qui le suit. |
| void remove() | Supprime l’objet qui vient d’être obtenu par *next*. Cette méthode est facultative. |

L'interface ListIterator dérive de l'interface Iterator

|  |  |
| --- | --- |
| int  nextIndex() | Retourne l'indice de l'élément qui serait retourné par un appel de next(). |
| void set(E o) | Remplace le dernier élément  retourné par next(), ou par previous(), par l'objet o. |
| void add(E o) | Remplace le dernier élément  retourné par next(), ou par previous(), par l'objet o. |