UEF4.3. Programmation Orientée Objet

COTOLE INTERMEDIAIRE (2h- Documents interdits)

Questions (4 pts)

1) Quelles sont les différences entre les exceptions contrôlées et les exceptions non contrôlées ? Donnez des exemples.

Exception contrôlées	Exceptions non contrôlées
- Un programme java bien écrit qui contient une partie du code pouvant générer (ou provoquer) une exception contrôlée **doit essayer de la traiter de l'une des deux manières suivantes sinon une erreur est signalée à la compilation 0.25**: - En la traitant avec le bloc try\catch - En la propageant avec le mot clé throws	Le compilateur ne contrôle pas ce type d'exceptions. **0.25**
Ce sont des sous classes de la classe Exception, elles peuvent être standard ou personnalisées **0.25**	Ce sont les exceptions de type **Error** ou de type **RuntimeException** **0.25**
Tout exemple de type Exception ou personnalisé est accépté **0.25**	Tout exemple de type type Error ou RuntimeException est accépté **0.25**

2) Citez les différents types de classes internes en précisant pour chaque type les droits d'accès aux membres de sa clase englobante. **(0.25+0.25)*4**

a. Les classes internes simples (non statiques) : elle accède aux membres de la classe englobante même s'ils sont privés
b. Les classes internes statiques : elles n'ont accès à aucun membre de la clase englobante, sauf s'il s'agit de membres statiques.
c. Les classes locales : Elles ont les mêmes droits d'accès qu'une classe interne non statique. En plus, **elles ne peuvent accéder qu'aux variables finales de la méthode dans laquelle elles sont définies**.
d. Les classes anonymes : Mêmes droits d'accès qu'une classe interne simple

3) Quels sont les cas où un constructeur sans arguments dans une classe dérivée lève une erreur.

- Si la classe mère possède un constructeur avec arguments et ne possède pas de constructeur sans arguments. **0.5**

Exercice 1 (4 points)

Qu'affichent les programmes suivants ? Donnez la bonne réponse. S'il y a une (des) erreur(s) de compilation ou d'exécution, expliquez la(les) cause(s) puis donnez la correction et le résultat de l'exécution après correction.

A/

```
class A {
public void f(){
System.out.print(" A:f "); }
public void f (int n) {
System.out.print(" A:int:"+n); }
public void f (int n, int q) {
System.out.print(" A:2int:"+q); }
public void f (int n, double y) {
System.out.print(" A:int-double:"+y);}
}
public class B extends A{
public void f () {
System.out.print(" B:f"); }
public void f (int n) {
System.out.print(" B:int:"+n); }
public void f (int n, int q) {
final int j=1;
class I{
public int f(int i){
System.out.print(" I:int:"+(i+j));
return (i+j);}
}
I i = newI();
System.out.print
(" B:2int:"+(q+i.f(n)));
}
public static void main(String args[]){
A a = newA(); a.f(); a.f(0);
a=new B(); a.f(); a.f(1);
a.f(1,1); a.f(3, 4.5);
}
}
```

a. Ce code comporte une erreur qui sera signalée à la compilation
b. Affiche un résultat que vous devez donner
c. Erreur à l'exécution

Réponse b : affiche un résultat que vous devez donner **0.5**
A:f A:int:0 B:f B:int:1 I:int:2 B:2int:3 A:int-double:4.5 **1.5**
-0.5 pour chaque erreur

B/

```
Package prog1;
public class A{
protected int i=0;
void f(int i) {
System.out.print(i);}
}
```

```
package prog2 ;
import prog1.A;
public class B extends A{
final int i=1;
public void f() {
System.out.print(i+" " +super.i);
super.f(this.i) ;}
public static void main(String args[]){
A a = new B(); a.f(2018); }
}
```

a. Ce code affiche 2018
b. Ce code affiche 1 0
c. Ce code affiche 0
d. Ce code comporte une erreur qui sera signalée à la compilation

La bonne réponse est d : « erreur à la compilation » **0.5**
Explication : la méthode f de A n'est pas public donc invisible en dehors d package **0.5**
Correction : rendre f de A public **0.5**
Résultat : 2018 **0.5**

Exercice 2 (5 pts)

A. Donner le code java des classes et interfaces suivantes :

1) L'interface `AugmentationSalaire` possède une méthode `double augmenterSalaire(double salaire, double pourcentage)` qui augmente par défaut le salaire de 10% qui lance une exception de type `AugmentationImpossibleException` si le nouveau salaire dépasse un plafond de 200000 DA
2) La classe `Fonctionnaire` implémente l'interface `AugmentationSalaire` et comporte:
   - Trois attributs: nom, prénom et salaire visibles par les classes dérivées
   - Un constructeur exhaustif (qui initialise tous ses attributs)
   - Une redéfinition de la méthode `equals` de façon à ce que deux fonctionnaires soient considérés égaux s'ils ont le même nom et le même prénom.
3) La classe `ChefDeService` hérite de la classe `Fonctionnaire` et possède:
   - Un attribut `Service` visible seulement par les classes se trouvant dans le même package
   - Un constructeur exhaustif.
   - Une surdéfinition de la méthode `augmenterSalaire()` de façon à l'augmenter de 15%.
4) La classe `Stagiaire` hérite de la classe `Fonctionnaire` et possède:
   - Un attribut `dureeDuStage` visible partout
   - Un constructeur exhaustif
   - Une redéfinition de la méthode `augmenterSalaire()` qui lance une exception de type `AugmentationImpossibleException` car un stagiaire ne peut pas bénéficier d'une augmentation.

B. Etant donné qu'un stagiaire n'a pas droit à une augmentation de salaire, comment pourrions nous corriger la conception proposée ?

Solution

A/ **(4 points)**

```
interface AugmentationSalaire{
   final double SALAIRE_MAX = 200000;
   public default  //0.25 double augmenterSalaire(double salaire, double pourcentage)
throws  AugmentationImpossibleException //0.5
{
      double newSalaire =   salaire +salaire*pourcentage/100; //0.25
//accepter aussi une solution qui utilise un taux au lieu d'un pourcentage
      if (newSalaire > SALAIRE_MAX) throw new AugmentationImpossibleException();
      else return newSalaire;
   }
}
```

Remarque : Si l'étudiant donne le throws sans le throw ou vice-versa il perd le 0.5

```
class Fonctionnaire implements AugmentationSalaire //0.25{

protected String nom, prenom;
protected double salaire; //0.25 pour les trois attributs, une erreur entraine la perte du 0.25. . Aucun modificateur d'accès hormis protected n'est accepté
```

```
public Fonctionnaire(String n, String p, double s){
    nom = n;
    prenom = p;
    salaire = s;
}
```
//0.25 pour le constructeur entier. Vérifier l'utilisation du mot cl this si l'étudiant utilise pour les arguments les même noms que les attributs

```
// le 0.25 n'est accordé que si la method est correcte
public boolean equals(Object f){
   if (((Fonctionnaire)f).nom.equals(this.nom)&&((Fonctionnaire)f).prenom.equals(this.prenom))return true ;
            else return false;
```
//0.25 si la méthode est correcte entièrement. Il faut que le type de l'argument soit Object sinon l'étudiant perd le 0.25

```
}

public double getSalaire(){
return salaire;
}
// il n'y a pas de points pour le getter il est nécessaire pour être utilisé dans le main ce qui n'est pas demandé
}


class ChefDeService extends Fonctionnaire
```
//0.25
```
{
    String service;
```
//0.25. Aucun modificateur d'accès n'est accepté

```
    public ChefDeService(String n, String p, double s, String service){
        super(n,p,s);
        this.service = service;
    }
```
//0.25 si le constructer est correct. L'étudiant perd le point s'l n'utilise pas super et si celui-ci n'est pas placé en premier

```
     public double augmenterSalaire(){
```
//0.25 si toute la méthode est correcte

```
        return  salaire +salaire*0.15;
    }
}

class Stagiaire extends Fonctionnaire { 
```
//0.25

```
    public int dureeDeStage ;
```
//0.25. Aucun modificateur hormis public d'accès n'est accepté

```
    public Stagiaire (String n, String p, double s, int d){
        super(n,p,s);
        dureeDeStage = d;
    }
```
//0.25 si le constructer est correct. L'étudiant perd le point s'l n'utilise pas super et si celui-ci n'est pas placé en premier

```
    public double augmenterSalaire(double salaire, double pourcentage) throws
AugmentationImpossibleException{
        throw new AugmentationImpossibleException();       }  }
```
//0.5 si la méthode est correcte entièrement. Si l'étudiant ne met pas Throw ou throws il perd le 0.5

class AugmentationImpossibleException extends Exception{}
// Cette classe n'est pas explicitement demandée dans l'énoncé. Attribuer 0.25 de bonus si l'étudiant l'ajoute

B/ Etant donné qu'un stagiaire n'a pas droit à une augmentation de salaire, la classe Stagiaire ne devrait pas descendre de la classe Fonctionnaire. Une meilleure solution consiste à la concevoir comme étant une classe séparée de la classe fonctionnaire. On peut éventuellement ajouter une classe mère (Employé par exemple) dont hérite Stagiaire et Fonctionnaire et qui contiendrait les attributs en commun : nom, prenom et salaire
// **(0.75 point) si l'étudiant dit que le lien d'héritage entre Stagiaire et Fonctionnaire ne doit pas exister**

Exercice 3 (6 points)

Nous désirons écrire un programme java permettant de vérifier à l'aide d'une pile si une expression arithmétique est correctement parenthésée.

1. Soit la classe Pile en figure 1 ci-dessous implémentant une pile en java. Compléter les lignes 5, 9, 13, 14, 15, 19.
2. Ecrire une classe java nommée ExpressionParenthesee ayant comme attribut une chaine de caractères représentant une expression arithmétique et nommée `expression`. Pour initialiser ce champ, on utilise un constructeur qui reçoit en entrée une chaine de caractères et utilise une pile pour vérifier si elle est correctement parenthésée. Si l'expression est erronée, le constructeur de la classe ExpressionParenthesee doit lancer une exception correspondant au type de l'erreur qui peut être due à une parenthèse ouvrante manquante ou une parenthèse fermante manquante comme le montrent les exemples suivants:

a+b	correcte
(a*b)+ (c*d)	correcte
(a+b)*c))	Parenthèse ouvrante manquante
)a+b(	Parenthèse ouvrante manquante
((a+b)*c)	Parenthèse fermante manquante

N.B: Nous considérons qu'en dehors des parenthèses, le reste de l'expression est correct.

3. Ecrire une classe nommée Exercice qui reçoit une chaine de caractères en argument de la ligne de commande et vérifie si elle est correctement parenthésée à l'aide de la classe ExpressionParenthesee. Vous pouvez vous aider de la méthode `charAt (int i)` de la classe `String` qui retourne le caractère qui se trouve à la $i^{ème}$ position en sachant que 0 est l'indice du premier caractère.

```
Import java.util.*;

class Pile {

     private Deque<Character>pile=new .....<Character>();       //ligne 5

     //  empile x au sommet de la pile
     public void empiler(char x){
          ..............................                        //ligne 9
     }
/*retourne l'élément qui est au sommet de la pile en le supprimant de la pile. Elle lance une
exception si la pile est vide*/
     public char depiler() ..............................{       //ligne 13
          if (this.estVide()) ..........................          //ligne 14
          ..............................................          //ligne 15

     }
     //retourne true si la pile est vide et false dans le cas contraire
     public boolean estVide() {
          ..............................                        //ligne 19
     }
}

class PileVideException extends Exception{}
```

figure 1: code java des classe Pile et PileVideException

Solution :
1/ 1.75

Ligne 5 **0,25**	`// Collection utilisée: ArrayDeque ou bien LinkedList` `privateDeque<Character>pile=newArrayDeque<Character>();`
Ligne 9 **0,25**	pile.addFirst(x) // ou pile.push(x)
Ligne 13 **0,5**	`// utilisation correcte des mots clé throws (dans l'entête) et throw dans le corps de la méthode` `public int depiler() throwsPileVideException{`
Ligne 14 **0,25**	`if (this.estVide()) throw new PileVideException();`
Ligne 15 **0.25**	`return` pile.removeFirst(); //ou `return` pile.pop();
Ligne 19 **0,25**	`return pile.isEmpty ();`

2. **/3,5**

Corrigé :

0,25	`classParentheseOuvranteExceptionextendsException{}`
0,25	`classParentheseFeranteExceptionextendsException{}`
0,25 **0,75**	`classExpressionParenthesee {` `private String expression;` **// La classe doit implémenter des deux types d'exception (0,25*2)+ le paramètre exp du constructeur (0,25)** `publicExpressionParenthesee(String exp)` `throwsParentheseOuvranteException,` `ParentheseFermanteException{`
0,25 **0,75**	**// Le programme doit utiliser une pile:** `Pile pile = newPile();` `int i = 0;` **/*empiler les parenthèses ouvrantes (0,25) et dépiler à la rencontre d'une parenthèse fermante (0,25) + la boucle (0,25) */** `try{while (i<exp.length()){` `char c = exp.charAt(i);` `if (c == '(')` `pile.empiler(newCharacter(c));` `else` `if(c == ')')` `pile.depiler();`

	i++;}
0,25	**/* Une expression est correcte si à la fin de la lecture des caractères de la chaine la pile est vide. Dans ce cas la chaine est affectée à l'attribut expression*/** `if (pile.estVide()) {expression = exp;` `System.out.println("expression correcte !");} // pas nécessaire`
0,25	**/*Si à la fin de la lecture de la chaine la pile n'est pas vide, il manque une parenthèse fermante*/** `else throw new ParentheseFermanteException();}`
0,25 **0,25** pr le try-catch	**// Le programme détecte une parenthèse ouvrante manquante si à la rencontre d'une parenthèse fermante la pile est vide** `catch(PileVideException e) { throw newParentheseOuvranteException(); }}}`

Ou bien (autre code) :

```
for (int i=0 ; i<exp.length() ; i++){
char c = exp.charAt(i);
if (c == '(') pile.empiler(newCharacter(c));
elseif(c == ')')
     try{pile.depiler(); }
catch(PileVideException e) { throw new ParentheseOuvranteException(); }
    }
if ( !pile.estVide()) {throw new ParentheseFermanteException();}
expression = exp;
System.out.println("expression correcte !");}
```

// Remarque : Ces solutions ne sont pas uniques. Toute autre solution correcte est acceptée

3. Ecrire une classe nommée Exercice2 qui reçoit une chaine de caractères en argument de la ligne de commande et vérifie si elle est correctement parenthésée à l'aide de la classe ExpressionParenthesee.

Remarque : Pour accéder au $i^{ème}$ caractère d'une chaine de caractère, nous pouvons utiliser la méthode `charAt(i)` de la classe `String`

Corrigé **1.75pt**

```
public class Exercice {
public static void main (String args[]){ 0,25
/* l'appel au constructeur de Expression parenthesee doit se faire dans un bloc try ayant deux blocs Catch pour chaque type d'erreur */
        try{0,25
/* Utiliation de args car l'énoncé précise que la chaine est passée en argument //de la ligne de commande*/
ExpressionParenthesee exp = new ExpressionParenthesee(args[0]); 0.25
             } catch (ParentheseOuvranteException e){ 0,5
```

```
            System.out.println("Expression mal parenthésée: parenthèse ouvrante manquante
!");
            }catch (ParentheseFermanteException e){ 0,5
                System.out.println("Expression mal parenthésée: parenthèse fermante
manquante !");
            }
}
}
```