Exercices POO suite (Java)

Exercice 1 (lecture de programmes)

1°) Les exemples suivants sont ils corrects ? Justifier.

```
class ClassA {
    int a;
    ClassA() {
        System.out.println(a);
    }

    ClassA(int a) {
        this.a = a;
        this();
    }
}
```

```
class ClassB {
    int a ;
}

class ClassBChild extends ClassB {
   int b ;
    ClassBChild(int a, int b) {
        this.a = a ;
        this.b = b ;
    }
```

```
}
```

```
class ClassC {
    int a ;
    ClassC(int a) {
       this.a = a ;
    }
}


class ClassCChild extends ClassC {
    int b ;
    ClassCChild(int a, int b) {
        this.a = a ;
        this.b = b ;
    }
}
```

```
class ClassD {
    public int a;
    ClassD() { }
    ClassD(int a) {
        this.a = a;
    }
}
```

```
class ClassDChild extends ClassD {
    private int a;
    int b;
    ClassDChild() {
        b = 10;
    }
    ClassDChild(int a, int b) {
        this();
        super();
    }
}
```

```
class ClassE {
    public int a;
    ClassE(int a) {
        this.a = a;
    }

    void setA(int c) {
        a = c;
    }
}

class ClassEChild extends ClassE {
    private int a;
```

```
    private ClassEChild(int a) {
        super(a);
        System.out.println("Constructor ");
    }

    private void setA(int c) {
        a = 2 * c;
    }
}
```

2°) Soient les classes suivantes:

```
import java.io.IOException;

class AThing {
   public int x;

   public AThing(int a) {
      System.out.println("Begin constructor AThing");
      x = a;
      System.out.println("AThing, x = " + x);
      System.out.println("End constructor AThing");
   }

   public void delete() {
      System.out.println("Begin delete AThing, x = " + x);
      System.out.println("End delete AThing");
   }
}

class BThing {
   public AThing aThing;
   public int y;



   public BThing(int a, int b) {
      System.out.println("Begin constructor BThing");
      y = b;
      System.out.println("BThing, y = " + y);
      aThing = new AThing(a);
      System.out.println("End constructor BThing");
   }

   public void delete() {
      System.out.println("Begin delete BThing, y = " + y);
      aThing.delete();
      System.out.println("End delete BThing");
   }
}

class CThing extends BThing {
   int z;

   CThing(int a, int b, int c) {
      super(a, a + b);
      System.out.println("Begin constructor CThing");
      z = c;
      System.out.println("CThing, z = " + z);
      System.out.println("End constructor CThing");
   }
```

```
    public void delete() {
        System.out.println("Begin delete CThing, z = " + z);
        System.out.println("End delete CThing");
        super.delete();
    }
}

public class Example {
    public static void main(String argv[]) throws IOException {
        System.out.println("Begin main allocation");
        BThing bThing = new BThing(1, 2);
        CThing cThing = new CThing(3, 4, 5);
        System.out.println("End main allocation");
        System.out.println("Begin main delete");
        cThing.delete();
        cThing = null;
        bThing.delete();
        cThing = null;
        System.out.println("End main delete");
    }
}
```

a) Quel est l'affichage du programme ?.

b) On ajoute la méthode print() dans la classe BThing

```
public void print() {
    System.out.println("aThing.x = " + aThing.x + " y = " + y);
}
```

et on modifie le main :

```
System.out.println("End main allocation");
System.out.println("bThing = ");
bThing.print();
System.out.println("cThing = ");
cThing.print();
System.out.println("Begin main delete");
```

Donner le nouvel affichage produit ?

c) On ajoute la méthode print() dans la classe cThing :

```
public void print() {
    System.out.println("aThing.x = " + aThing.x + " y = " + y + " z = " + z);
}
```

Donner le nouvel affichage produit ?

Exercice 2

On souhaite disposer d'un ensemble de classes permettant de manipuler des formes géométriques en 2 et 3 dimensions (on ne s'occupe pas de leur représentation graphique, il est juste question de manipuler convenablement la technique de l'héritage).

a) Architecture des classes

Ecrire les classes reflétant la structure suivante (chaque indentation désigne un héritage) :

- Geometry (**)
  - Geometry2D (*)
    - Rectangle
      - Square
    - Ellipsis
      - Circle
  - Geometry3D (*)
    - Sphere
    - Cylinder

(*) classe abstraite, définissant une fonction d'affichage
(**) interface ou classe abstraite base de la hiérarchie

Ces classes doivent permettre les tâches suivantes :

- identifier les objets lors de leur création (Circle c = new Circle("Head", 2.5))
- calculer la surface des objets (méthode area())
- calculer le périmètre des objets dérivant de Geometry2D (méthode perimeter())
- calculer le volume des objets dérivant de Geometrie3D (méthode volume())
- comparer deux objets par exemple sur leur surface ou leur volume

b) Ecrire un petit programme Java mettant en oeuvre ces classes. En particulier, construire un tableau de Geometry2D et le trier.

c) Déployer ces classes dans les packages :

- shapes
- shapes.shapes2D
- shapes.shapes3D

d) Ajouter des variables de classes comptabilisant le nombre d'instances des classes Geometry, Geometry2D et Geometry3D (prévoir le cas de suppression d'instances).

e) Génération de la documentation

Insérer des commentaires conformes aux spécifications de l'utilitaire javadoc et générer la documentation HTML des packages avec cet outil (bien noter que seuls les membres publics et protégés sont commentés).

Rappels :

Formes	**Surface**	**Périmètre**	**Volume**
Rectangle	l * h	2(l+h)	
Carré	l^2	4l	
Cercle	πr^2	π 2r	
Ellipse	π ab		
Sphère	$4\pi r^2$		
Cylindre	π 2rh		$\pi r^2 h$

Notes :

- l = largeur
- h = hauteur
- r = rayon
- a = rayon maximum (du grand axe
- b = rayon minimum (du petit axe)

Exercice 3

Pour ceux qui auraient encore des problèmes avec l'héritage, voici un exercice supplémentaire simple. Il s'agit dans ce problème de créer une série de classes, reliées par héritage.

a) la classe Building
La classe est formée des membres suivants :

- owner : c'est le nom du propriétaire du bâtiment,
- address : c'est l'adresse du bâtiment,
- area : c'est la surface (en m^2) qu'occupe le bâtiment (terrain compris)

Il y a deux fonctionnalités générales pour un bâtiment (donnant lieu à deux méthodes) :

- tax: cette méthode retourne une valeur entière et dépend de la nature du bâtiment,
- print

Y rajouter toutes les méthodes nécessaires.

b) la classe Villa
Il s'agit d'une dérivée de Bâtiment. Elle contient en outre deux autres caractéristiques :

- roomCounter : nombre de pièces
- swimming-pool : c'est un booléen qui indique la présence ou non d'une piscine privée.

L'impôt vaut 100€ par pièces + 750€ si jamais il y a une piscine.

c) la classe Company
Entreprise est aussi une dérivée de la classe Bâtiment et possède en plus:

- name : un nom
- employeeCounter : le nombre de salariés,
- avg : un nombre moyen de livraisons journalières à l'usine.

L'impôt local sur une entreprise se calcule de la façon suivante : 6.3€/m2.
Construire et tester ces classes. Enfin rajouter un main permettant de regrouper dans un tableau statique (obligatoirement) plusieurs bâtiments quelconques et effectuant un certain nombre de manipulations en correspondance.
En particulier, permettre :

- le calcul de la surface totale occupée par tous les bâtiments stockés dans le tableau
- le calcul de l'impôt global dû par tous les bâtiments stockés dans le tableau
- le nombre de piscine
- le déclenchement de plusieurs tris du tableau :
    - un sur la surface
    - un sur le nom du propriétaire (étudier l'interface Comparator et créer un objet anonyme).

Exercice 4

On souhaite dans cet exercice modéliser le fonctionnement d'un système de fichiers. On suppose donc qu'une arborescence commence par un répertoire unique et que celui-ci peut contenir d'autres répertoires et d'autres fichiers. Chaque répertoire peut à son tour posséder répertoires et fichiers.
Pour simplifier, supposer que les données nécessaires sont simplement un nom, une taille et une date de création.
Construire une application permettant de :

- gérer une arborescence
- rajouter un répertoire ou un fichier
- supprimer un répertoire ou un fichier
- lister un répertoire
- calculer la taille totale d'un répertoire.

Exercice 5

Cet exercice traite des mêmes problématiques que le précédent, mais gère une liste chaînée en interne.
Compléter les classes suivantes et définir les classes dérivées Person, Employee, Worker et Exercise.

```
public class Person {
    String nom;
    ...
}

abstract public class Employee {
    Person person;
    float salary;
    int service;
    protected Employee next;
    private static Employee salaryList = null;

    public Employee(float salary, int service) { ... }

    public Employee(String name, float salary, int service) { ... }

    public static void printAll() { ... }

    public void deleteAll() { ... }

    public void delete() { ... }

    abstract void print();
}

public class Worker extends Employe {
    int hours;
    ...
    }
```

```
public class Exercise {
```

```
    public static void main(String args[]) {
      Worker r1 = new Worker ("Mathieu",1500,1,169);
      Worker r2 = new Worker ("Valentin",1500,2,182);
      Worker r3 = new Worker ("Vincent",1500,3,167);
      Employee.printAll();
      r3.delete();
      Employee.printAll();
      r3.deleteAll();
      Employee.printAll();
    }
}
```

Exercice 6

L'objectif de cet exercice est de réfléchir aux annotations, et à l'introspection.

a) Construire un type annotation, de nom @Notice, possédant un paramètre value de type int, de valeur par défaut 1. Construire ensuite une classe utilisant cette annotation, puis une application affichant la liste des champs de cette même classe, accompagnés des valeurs annotées.

b) Modifier le programme précédent de la manière suivante : afficher l'intégralité des champs et méthodes annotés par une annotation @Notice dont le paramètre value est supérieur ou égal à value.

c) Déclarer une annotation NoticePrint tel que pour un objet :
   - si un champs est marqué NoticePrint ou NoticePrint (true) alors le champs est affiché si c'est un type primitif ou String
   - si un champs est marqué NoticePrint ou NoticePrint (true) et si son type est autre alors l'ensemble des champs de l'objet est récursivement affiché.
   - si un champs est marqué NoticePrint (false) ou n'est pas marqué par NoticePrint alors le champs n'est ni affiché ni parcouru récursivement.

   L'annotation ne doit être positionnable que sur les champs.

d) Écrire une fonction deepNoticePrint(Object o) qui affiche récursivement les champs de l'objet en utilisant l'annotation NoticePrint.

Résumé Annotation

Les annotations sont un outil formidable permettant de parsemer le code Java de méta-informations, c'est-à-dire d'instructions qui ne sont pas destinées à produire du code machine, mais à produire de l'information pour des parsers extérieurs. Il est possible de comparer les annotations aux directives du préprocesseur C, mais il y a des différences :

Directives du préprocesseur C	**Annotations Java**
Sont traitées uniquement avant la compilation et modifient le code source du fichier	Sont traitées avant la compilation et peuvent produire des fichiers sources
	Sont traitées à la compilation, et peuvent produire des informations à destination du développeur
	Sont traitée à l'exécution d'un programme pour en modifier le comportement

Le mécanisme, introduit par Java 5, est une extension des méta-informations utilisées par la commande Javadoc.

```
/**
 * Méthode de traitement général
 * @param value : paramètre de définition
 * @return : calcul effectué par la fonction
 * @throws IOException : émet une IOException si le fichier n'existe pas
 **/
 String f(int value) throws IOException {
   ...
}
```

I. PRESENTATION DES ANNOTATIONS

1°) Généralités

Depuis Java 5, il s'agit de l'une des nouveautés les plus intéressantes de Java 5.0, permettant de transférer une partie du travail de codage du programmeur au compilateur (et donc de simplifier la programmation).

> **definition** : les annotations sont des informations, appelées méta-informations, qu'il est possible d'ajouter au code source, récupérables et exploitables soit par des outils de pré-compilation, soit par le compilateur, soit à l'exécution d'une application Java par les outils standards d'introspection et de réflexion

Ces méta-informations peuvent être appliquées à des déclarations de package, de déclarations de types, de constructeurs, de méthodes, de données membres, de paramètres de fonction ou à des variables

On peut donc comparer par exemple les annotation à des instructions lisibles par le préprocesseur C (il existait d'ailleurs à la version 5 un outil équivalent appelé APT, sorte de javac de prétraitement, maintenant remplacé par une option de la commande javac), ou encore aux commentaires exploitables par l'outil javadoc.

Il s'agit d'un instrument extrêmement puissant permettant par exemple :

- **d'indiquer si certaines méthodes dépendent d'autres méthodes,**
- **d'indiquer si elles sont complètes ou non,**
- **d'indiquer si les classes font référence à d'autres classes,**
- **de générer des fichiers XML à la compilation,**
- **de générer des classes à la compilation (cas des services Web),**
- **de fournir des informations au compilateur, de manière à détecter certaines erreurs ou supprimer des warnings**
- **de servir à des logiciels extérieurs, de manière à générer du code, des fichiers XML ou autres**
- **d'être analysées au moment de l'exécution,**
- **de servir à mettre en place des tests unitaires**

En bref, tout ce qui est injection de code peut être programmé à l'aide d'une annotation.

2°) Utilisation d'une annotation
L'utilisation d'une annotation se fait de manière simple à l'aide du symbole @.

Syntaxe:

@NomAnnotation[(p1="v1",p2="v2", …, pn="vn")]

Ce marqueur peut être accompagné de paramètres (ici donné dans la syntaxe sous la forme p_i=v_i).

Par exemple :

```
@Override
@SuppressWarnings(value = "unchecked")
@SuppressWarnings(value = {"rawtypes","unused"} )
```

Les annotations peuvent être utilisées sur à peu près tous les éléments de la programmation objet Java :

- les classes, interfaces et énumérations,

- les données membres ou les méthodes de classe,
  ```
  @ToDo
  public void methode() {
  }
  ```

- les paramètres des méthodes,

- les variables locales,
  ```
  @Deprecated
  public class Example {
      …

      @Deprecated
      public int compute(@Deprecated int _nb) {
          @Deprecated
          int i;
      }
  }
  ```

- les packages, mais il faut construire un fichier particulier, appelé package-info.java, contenant la déclaration du package, de sa documentation et de ses annotations
  ```
  /** Exemple de définition de package avec annotations */
  @Usefull
  package fs.esgi.java;
  ```

- d'autres annotations ...

Un des points les plus intéressants est que les annotations que l'on peut utiliser proviennent :

- soit du langage lui-même, qui à chaque version en rajoute un certain nombre comme par exemple @Override, @SuppressWarnings, @Deprecated ...
- soit des APIs (les services Web en font grand usage),

- soit des développeurs eux-mêmes; en effet, chaque programmeur est libre d'inventer ses propres annotations. Il existe en effet la possibilité de définir ses propres types d'annotation (une sorte de classe modélisant une annotation)

3°) Lesprincipales annotations fournies par Java

Il en existe plusieurs qui sont fournies par défaut, mais trois importantes :

- Override : indique qu'une méthode est construite pour redéfinir une méthode d'une classe de base; si une méthode annotée par Override ne le fait pas, le compilateur génère une erreur (ne fonctionne que pour redéfinir des méthodes existantes et non implémenter des méthodes abstraites ou d'interfaces).

```
class Base {
	public int f() {
		return 1;
	}
}

class Derivee extends Base {
	@Override
	public int f() {
		return super.f()+10;
	}
}
```

- Deprecated : indique qu'un élément de programmation déprécié est défini. Elle doit s'accompagner d'un texte expliquant pourquoi une méthode est dépréciée et par quoi la remplacer

```
 /**
   * utilisation de@deprecated doSomething est déconseillée, utiliser
   * doSomethingElse à la place
   */

class TestDeprecated {
	  @Deprecated
	  public void doSomething() {
	    System.out.println("Test d'annotation dépréciée pour une méthode");
	  }
	}

public class TestInt {

	public void print() {
		TestDeprecated der=new TestDeprecated ();
		der.doSomething();
	}

	public static void main(String[] args) {
		TestInt ti = new TestInt();
		ti.print();
	}
}
```

L'utilisation provoquera de cette méthode provoquera un avertissement :

```
C:\Users\jeff\workspfred2010mai\TestGraphique\src>javac TestInt.java
Note: TestInt.java uses or overrides a deprecated API.
```

- SuppressWarnings("nommessage") : indique que les messages d'avertissement du compilateur ne doivent pas être affichés .

```
public class TestInt {

	@SuppressWarnings("deprecation")
	public void print() {
		TestDeprecated der=new TestDeprecated ();
		der.doSomething();
	}

	public static void main(String[] args) {
		TestInt ti = new TestInt();
		ti.print();
	}
}
```

ne provoquera pas d'affichage d'avertissement à la compilation

II. LES TYPES ANNOTATIONS

1°) Les familles d'annotation

Il existe trois types d'annotation :

- **les annotations de type "marqueur" : les marqueurs sont des annotations vides, sans paramètres à l'exception du nom de l'annotation elle-même**

```
public @interface NomAnnotation {
}

@NomAnnotation
public void methode() {
  …
}
```

- **les annotations de type éléments simples ou paramétrées : elles permettent l'utilisation d'un paramètre, représentable sous la forme d'une seule propriété de forme (donnée**="valeur") ou en abrégé ("valeur") si le paramètre est nommé value

```
public @interface NomAnnotation {
 . String value();
}

@NomAnnotation("Valeur")
public void methode() {
  …
}
```

- **les annotations "Full-value" ou multi-paramétrées :** il existe dans ce cas plusieurs paramètres, d'utilisation donnée=valeur pour chaque membre.

```
public @interface NomAnnotation {
  String parameter();
  int count();
  String date();
}

@MonAnnotation(parameter="Valeur", count=1, date="21/01/2010")
public void methode() {
  …
}
```

Il est nécessaire de respecter les règles suivantes lors de la définition de types annotation :

1. les déclarations de type doivent commencer par @interface, suivi par le nom de l'annotation,
2. Les déclarations de méthode ne doivent pas avoir de paramètres (String parameter() ne peut pas recevoir de paramètre)
3. Les déclarations de méthode ne doivent pas avoir de clause throws
4. les types de valeurs de retour des méthodes sont :
   - primitives
   - String
   - Class
   - enum
   - tableau des types précédents

2°) La fabrication d'un type annotation

Il est donc possible de fabriquer son propre type d'annotation (une sorte de classe modélisant une annotation).

Syntaxe :

```
public @interface NomAnnotation {
  // Définition des paramètres
  type parametre() default ["valeur par défaut"];
   …
}
```

Exemple :

```
public @interface MyAnnoation {
  . String name();
}
```

Les paramètres doivent être déclarés dans le corps de la définition du type annotation.

Remarques :

- Les parenthèses doivent être interprétées comme une sorte de définition de méthode d'accès au paramètre dont le nom est "nom". Les types des paramètres sont limités : types primitifs, String, annotation, énumération, Class ou tableaux à une dimension

- Il peut y avoir des valeurs par défaut :

```
public @interface GroupTODO {
  public enum Severity { CRITICAL, IMPORTANT, TRIVIAL, DOCUMENTATION };

  Severity severity() default Severity.IMPORTANT;
  String item();
  String assignedTo();
  String dateAssigned();
}
```

3°) Les méta-annotations.

Il s'agit d'annotations utilisées pour annoter **des déclarations de types d'annotations**, elles sont appelées annotations d'annotations.

Il en existe 4 types :

- **l'annotation Target** : elle indique les informations sur lesquelless sont applicables une annotation. Si elle est absente, l'annotation peut être utilisée sur n'importe quoi sauf sur le type PARAMETER (en cas d'erreur, le compilateur préviendra):
  - @Target(ElementType.TYPE) applicable à n''importe quel element d'une classe
  - @Target(ElementType.FIELD) applicable à un champ/propriété
  - @Target(ElementType.METHOD)) applicable à une méthode
  - @Target(ElementType.PARAMETER)) applicable aux paramètres d'une méthode
  - @Target(ElementType.CONSTRUCTOR)) applicable aux constructeurs
  - @Target(ElementType.LOCAL_VARIABLE) applicable aux variables locales
  - @Target(ElementType.ANNOTATION_TYPE) indique que le type declare est lui-même une annotation

Tout ceci est défini dans :

```
package java.lang.annotation;

public enum ElementType {
 TYPE,                  // Class, interface, or enum (but not annotation)
 FIELD,                 // Field (including enumerated values)
 METHOD,                // Method (does not include constructors)
 PARAMETER,             // Method parameter
 CONSTRUCTOR,           // Constructor
 LOCAL_VARIABLE,        // Local variable or catch clause
 ANNOTATION_TYPE,       // Annotation Types (meta-annotations)
 PACKAGE                // Java package
}
```

Exemple :

```
import java.lang.annotation.Annotation;
import java.lang.annotation.Target;
import java.lang.annotation.ElementType;

 @Target(ElementType.METHOD)
 @interface Comment {
        String explication();
}

public class TestInt {
   @Commentaire(explication="fonction obligatoire")
        public void print() {
                System.out.println("fonction annotée");
        }

        public static void main(String[] args) {
                TestInt ti = new TestInt();
                ti.print();
        }
}
```

- **l'annotation Retention** : indique quand sont traitées les annotations :
    - **RetentionPolicy.SOURCE** : doivent être traitées au niveau du source et seront ignorées par le compilateur. Il faut dans ce cas construire un processeur d'annotation.
    - **RetentionPolicy.CLASS** : doivent être traitées au niveau du compilateur et seront ignorées par la JVM (valeur par défaut); attention, dans ce cas, elles sont ignorées lors de l'exécution du programme (et donc getAnnotations renvoie un tableau vide)
    - **RetentionPolicy.RUNTIME** : doivent être traitées par la JVM et ne peuvent donc être lues qu'au moment de l'exécution (par introspection).

```
@Retention(RetentionPolicy.RUNTIME)
public @interface Test_Retention {
  String doTestRetention();
}
```

Les valeurs sont définies de la manière suivante :

```
package java.lang.annotation;

public enum RetentionPolicy {
 SOURCE,          // Annotation is discarded by the compiler
 CLASS,           // Annotation is stored in the class file, but ignored by VM
 RUNTIME          // Annotation is stored in the class file and read by the VM
}
```

- **l'annotation Documented** : indique qu'une annotation de ce type doit être prise en compte par l'utilitaire javadoc pour la production de documentation (par défaut les annotations ne sont pas incluses dans javadoc)

```
@Documented
@Target(ElementType.METHOD)
@interface Comment {
        String explication();
}
```

- **l'annotation Inherited** : indique que l'annotation sera héritée par tous les descendants de l'élément sur lequel elle a été posée. Par défaut, les annotations ne sont pas héritées par les éléments fils.

```
public @interface SimpleAnnotation { }

import java.lang.annotation.Inherit;
@Inherited
public @interface InheritAnnotation { }

@SimpleAnnotation
@InheritAnnotation
public class ExampleInherited {
}
```

Toutes les classes qui étendront ExampleInherited seront alors automatiquement marquées par l'annotation @InheritAnnotation mais pas par @SimpleAnnotation...

III. L'UTILISATION DES ANNOTATIONS AU RUNTIME

Il est possible d'accéder aux annotations elles-mêmes par introspection de la manière suivante (ATTENTION UNIQUEMENT POUR LES RetentionPolicy.RUNTIME) :

```
import java.lang.annotation.Annotation;

Class c = NomClasse.class;
Annotation[] annotations = c.getAnnotations();

for(Annotation annotation : annotations){
  if(annotation instanceof NomAnnotation){
    NomAnnotation var = (NomAnnotation) annotation;
    System.out.println("nom: " + var.nom());
  }
}
```

ou de celle-ci

```
import java.lang.annotation.Annotation;

Class c = NomClasse.class;
Annotation annotation = aClass.getAnnotation(MyAnnotation.class);

if(annotation instanceof NomAnnotation){
  NomAnnotation var = (NomAnnotation) annotation;
  System.out.println("nom: " + var.nom());
}
```

Attention à la fonction getAnnotations qui doit être déclenchée sur l'élément sur lequel on déclenche la recherche d'annotation; pr exemple, si l'on souhaite les annotations apportées sur les méthodes d'une classe, il ne faut pas déclencher getAnnotations sur getClass mais sur getClass().getMethods()