# Correction TD2

## Partie 1 : Interfaces

### Exercice 1 :

#### Q 1.1. Code minimal pour la classe Elephant :

package zoo;  
   
public class Elephant implements Animal {  
 private int weight; // Poids en kilogrammes  
   
 public Elephant(int weight) {  
 this.weight = weight;  
 }  
   
 @Override  
 public int getNbLegs() {  
 return 4;  
 }  
   
 @Override  
 public boolean isCarnivorous() {  
 return false;  
 }  
}

#### Q 1.2. Réponse :

Non, il n’est pas nécessaire de compléter la classe Zoo pour compiler et exécuter les lignes de code fournies. La classe Zoo n’a pas été fournie dans l’énoncé, mais à condition qu’elle ait une méthode addAnimal(Animal animal) comme décrite dans l’énoncé, elle devrait être suffisante pour compiler et exécuter les lignes de code fournies.

#### Q 2. Possibles valeurs pour T :

La déclaration T ref = new Elephant(1000); indique que la référence ref peut être de tout type compatible avec Elephant, c’est-à-dire qu’elle peut être un type de classe ou une interface que Elephant implémente ou une superclasse de Elephant. Par conséquent, les valeurs possibles pour T sont : - Elephant - Animal - Object (car toutes les classes Java héritent implicitement de Object)

### Exercice 2 :

#### Q 1. Classe ToLowerCaseTransformation :

package text;  
   
public class ToLowerCaseTransformation implements Transformation {  
 @Override  
 public String transform(String input) {  
 return input.toLowerCase();  
 }  
   
 @Override  
 public String getDescription() {  
 return "Transforms all uppercase characters to lowercase.";  
 }  
}

#### Q 2. Valeurs possibles pour T :

La déclaration T ref = new ToLowerCaseTransformation(); indique que la référence ref peut être de tout type compatible avec ToLowerCaseTransformation, c’est-à-dire qu’elle peut être un type de classe ou une interface que ToLowerCaseTransformation implémente ou une superclasse de ToLowerCaseTransformation. Par conséquent, les valeurs possibles pour T sont : - ToLowerCaseTransformation - Transformation - Object

#### Q 3. Méthode multiTransform :

package text;  
   
public class MultiStringTransformer {  
 public static String[] multiTransform(String[] strings, Transformation transformation) {  
 String[] transformedStrings = new String[strings.length];  
 for (int i = 0; i < strings.length; i++) {  
 transformedStrings[i] = transformation.transform(strings[i]);  
 }  
 return transformedStrings;  
 }  
}

#### Q 3.1 Méthode multiTransform avec programmation fonctionnelle :

package text;  
   
import java.util.Arrays;  
public class MultiStringTransformer {  
 public static String[] multiTransform(String[] strings, Transformation transformation) {  
 return Arrays.stream(strings)  
 .map(transformation::transform)  
 .toArray(String[]::new);  
 }  
}

#### Q 4. Séquence de lignes de code :

// 1. Créer un objet MultiStringTransformer  
MultiStringTransformer transformer = new MultiStringTransformer();  
   
// 2. Définir une variable lowers avec pour valeur le tableau des chaînes de caractères transformées en minuscules  
String[] lowers = transformer.multiTransform(data, new ToLowerCaseTransformation());  
   
// 3. Définir une variable rot13s avec pour valeur le tableau des chaînes de caractères codées en Rot-13  
String[] rot13s = transformer.multiTransform(data, new Rot13Transformation());

Dans ce code, nous utilisons la méthode multiTransform de la classe MultiStringTransformer pour transformer le tableau de chaînes de caractères data en minuscules et en Rot-13, respectivement, en utilisant les transformations ToLowerCaseTransformation et Rot13Transformation.

## Partie 2 : Programmation fonctionnelle

### Exercice 1:

import java.util.Arrays;  
import java.util.List;  
import java.util.stream.Collectors;  
   
public class Main {  
 public static void main(String[] args) {  
 List<String> strings = Arrays.asList("Java", "Python", "JavaScript", "C", "Ruby");  
 List<Integer> lengths = strings.stream()  
 .map(String::length)  
 .collect(Collectors.toList());  
 System.out.println(lengths); // Affiche : [4, 6, 10, 1, 4]  
 }  
}

### Exercice 2:

import java.util.Arrays;  
import java.util.List;  
import java.util.stream.Collectors;  
   
public class Main {  
 public static void main(String[] args) {  
 List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);  
 List<Integer> squares = numbers.stream()  
 .map(n -> n \* n)  
 .collect(Collectors.toList());  
 System.out.println(squares); // Affiche : [1, 4, 9, 16, 25]  
 }  
}

### Exercice 3:

import java.util.Arrays;  
import java.util.List;  
import java.util.stream.Collectors;  
   
public class Main {  
 public static void main(String[] args) {  
 List<String> strings = Arrays.asList("Apple", "Banana", "Strawberry", "Orange", "Kiwi");  
 List<String> longStrings = strings.stream()  
 .filter(s -> s.length() > 5)  
 .collect(Collectors.toList());  
 System.out.println(longStrings); // Affiche : ["Strawberry", "Orange"]  
 }  
}

### Exercice 4:

import java.util.Arrays;  
import java.util.List;  
   
public class Main {  
 public static void main(String[] args) {  
 List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);  
 int sum = numbers.stream()  
 .reduce(0, Integer::sum);  
 System.out.println(sum); // Affiche : 15  
 }  
}

### Exercice 5:

import java.util.Arrays;  
import java.util.List;  
   
public class Main {  
 public static void main(String[] args) {  
 List<Integer> numbers = Arrays.asList(10, 20, 30, 40, 50);  
 double average = numbers.stream()  
 .mapToDouble(n -> n)  
 .average()  
 .orElse(0);  
 System.out.println(average); // Affiche : 30.0  
 }  
}

### Exercice 6:

import java.util.Arrays;  
import java.util.List;  
   
public class Main {  
 public static void main(String[] args) {  
 List<String> strings = Arrays.asList("Hello", "World", "Java", "Programming", "Language");  
 String concatenated = strings.stream()  
 .collect(Collectors.joining(", "));  
 System.out.println(concatenated); // Affiche : Hello, World, Java, Programming, Language  
 }  
}

### Exercice 7:

import java.util.Arrays;  
import java.util.List;  
import java.util.stream.Collectors;  
   
public class Main {  
 public static void main(String[] args) {  
 List<String> strings = Arrays.asList("apple", "banana", "orange", "kiwi", "strawberry");  
 List<String> upperStrings = strings.stream()  
 .map(String::toUpperCase)  
 .collect(Collectors.toList());  
 System.out.println(upperStrings); // Affiche : ["APPLE", "BANANA", "ORANGE", "KIWI", "STRAWBERRY"]  
 }  
}

### Exercice 8:

import java.util.Arrays;  
import java.util.List;  
import java.util.stream.Collectors;  
   
public class Main {  
 public static void main(String[] args) {  
 List<Integer> numbers = Arrays.asList(3, 8, 2, 5, 10, 7);  
 List<Integer> evenSortedNumbers = numbers.stream()  
 .filter(n -> n % 2 == 0)  
 .sorted()  
 .collect(Collectors.toList());  
 System.out.println(evenSortedNumbers); // Affiche : [2, 8, 10]  
 }  
}

### Partie 3 : Questions de cours

#### Exemple de classe :

interface Forme {  
 double calculerSurface();  
}  
   
interface Forme2D extends Forme {  
 double calculerPerimetre();  
}  
   
interface Forme3D extends Forme {  
 double calculerVolume();  
}  
   
class Rectangle implements Forme2D {  
 private double longueur;  
 private double largeur;  
   
 public Rectangle(double longueur, double largeur) {  
 this.longueur = longueur;  
 this.largeur = largeur;  
 }  
   
 @Override  
 public double calculerSurface() {  
 return longueur \* largeur;  
 }  
   
 @Override  
 public double calculerPerimetre() {  
 return 2 \* (longueur + largeur);  
 }  
}

### Réponses aux questions :

1. **Différence entre classe abstraite et interface :**
   * Une classe abstraite peut contenir des méthodes concrètes en plus des méthodes abstraites, tandis qu’une interface ne peut contenir que des méthodes abstraites.
   * Une classe peut étendre une seule classe abstraite, mais elle peut implémenter plusieurs interfaces.
   * Une classe abstraite peut avoir des champs (variables d’instance), tandis qu’une interface ne peut contenir que des constantes (champs static final) et des méthodes abstraites.
   * Une interface est utilisée pour définir un contrat commun pour plusieurs classes, tandis qu’une classe abstraite est utilisée pour définir des comportements communs pour ses sous-classes. Exemple :
   * Une classe abstraite Animal pourrait contenir des méthodes concrètes telles que manger() et dormir(), tandis qu’une interface Volant pourrait contenir une méthode abstraite voler() que plusieurs classes telles que Oiseau et Avion pourraient implémenter. **Pour autant un Oiseau et un Avion ne sont pas des animaux, donc l’interface est plus adaptée.**
2. **Qu’est-ce qu’une interface :**
   * Une interface est une collection de méthodes abstraites. Elle définit un contrat que les classes qui l’implémentent doivent respecter.
3. **Avantage de l’utilisation d’une interface par rapport à une classe abstraite :**
   * Les interfaces permettent une plus grande flexibilité, car une classe peut implémenter plusieurs interfaces, mais elle ne peut étendre qu’une seule classe abstraite.
   * Les interfaces permettent également de simuler l’héritage multiple, ce qui n’est pas possible avec les classes abstraites.
4. **Utilisation d’une interface pour définir un contrat commun :**
   * Une interface définit un ensemble de méthodes que les classes qui l’implémentent doivent fournir, garantissant ainsi un comportement commun.
5. **Déclaration de Forme2D et Forme3D comme des interfaces :**
   * Nous déclarons Forme2D et Forme3D comme des interfaces plutôt que des classes abstraites car cela permet aux classes de posséder d’autres fonctionnalités en plus de celles fournies par l’interface.
6. **Implémentation de plusieurs interfaces dans une classe :**
   * Oui, une classe peut implémenter plusieurs interfaces. Cela s’appelle une implémentation multiple.
   * Les interfaces permettent une implémentation multiple sans introduire de complexité supplémentaire comme c’est le cas avec l’héritage multiple des classes.
7. **Signification de implements Forme2D pour la classe Rectangle :**
   * Cela signifie que la classe Rectangle doit fournir une implémentation des méthodes définies dans l’interface Forme2D.
8. **Méthodes que la classe Rectangle doit implémenter en raison de son implémentation de l’interface Forme2D :**
   * calculerSurface() et calculerPerimetre(). Dans notre exemple, la classe Rectangle doit implémenter ces deux méthodes pour être considérée comme une Forme2D.