Nom, prénom :

Partiel de Compléments en Programmation Orientée Objet n° 2
(Correction)

*Pour chaque case, inscrivez soit "**V**"(rai) soit "**F**"(aux), ou bien ne répondez pas.*
Note = max(0, nombre de bonnes réponse - nombre de mauvaises réponses), ramenée au barême.
Sauf mention contraire, les questions concernent Java 8.

Rappel, quelques interfaces définies dans `java.util.function` :

```
public interface Consumer<T> { void accept(T t); }
```

```
public interface Function<T, R> { R accept(T t); }
```

Questions :

1. [V] Les attributs d'une interface sont tous statiques.

   Correction : Oui, par conception du langage. La raison est que si une interface avait des attributs d'instance, elle forcerait ses implémentations à contenir ces données, ce qui va au delà des prérogatives d'une interface (à savoir : définir les intéractions avec les objets et non pas leur mise en œuvre).

2. [F] Une classe implémentant une interface `I` doit (re)définir toutes les méthodes déclarées dans `I`.

   Correction : 2 raisons pour lesquelles c'est faux :
   - une classe abstraite peut implémenter une interface sans redéfinir toutes les méthodes déclarées (qui restent abstraites) ;
   - une interface peut contenir des méthodes non abstraites : `default`, qui n'ont pas à être redéfinies, et `static` pour lesquelles le concept-même de redéfinition est absurde.

3. [F] La méthode `somme` ci-dessous s'exécute toujours sans erreur (ne quitte pas sur une exception) :

```
import java.util.List; import java.util.ArrayList; import java.util.Collections;
public class PaquetDEntiers {
    private final List<Integer> contenu; private final int taille;
    public PaquetDEntiers(ArrayList<Integer> contenu) {
        if (contenu != null) this.contenu = contenu;
        else this.contenu = Collections.emptyList(); // initialisation à liste vide
        this.taille = this.contenu.size();
    }
    public int somme() {
        int s = 0; for (int i = 0; i < taille; i++) { s += contenu.get(i); } return s;
    }
}
```

   Correction : C'est un problème d'*aliasing* : si on initialise une instance de `PaquetDEntiers` avec une liste non vide, puis qu'on supprime un élément de la liste avant de demander à l'instance de `PaquetDEntiers` de calculer la somme, on aura `IndexOutOfBoundsException` : en effet, le nombre d'itérations pour calculer la somme est calé sur la taille qu'avait la liste au moment de la construction. Si la taille a été diminuée entre temps, le calcul de la somme va faire un appel à `get` sur un indice qui n'est plus dans la liste, d'où l'erreur.
   Pour rendre cette classe robuste, il faut initialiser l'attribut `contenu` avec une copie défensive du paramètre du constructeur.
   La solution consistant à supprimer l'attribut redondant `taille` et se servir de `contenu.size()` comme borne du `for` fonctionne aussi, mais seulement dans un contexte *single-threaded*. Dans un contexte *multi-thread*, en cas d'accès concurrents à la liste *aliasée*, il peut encore y avoir des soucis.

Cela dit, dans tous les cas de figure, c'est une bonne chose de supprimer les attributs redondants.

4. F Le code source doit être recompilé en code-octet avant chaque exécution.

   Correction : Le code-octet peut évidemment être ré-exécuté à volonté.

5. F Quand on "*cast*" (transtype) une expression d'un type référence vers un autre, dans certains cas, Java doit, à l'exécution, modifier l'objet référencé pour le convertir.

   Correction : Non. Le principe d'un *cast* d'objet, c'est "ça passe ou ça casse" : soit l'objet a le type demandé et on peut l'utiliser sans modification ; soit ce n'est pas le cas, et le programme quitte sur une exception (`ClassCastException`).

6. F La conversion de `long` vers `double` ne perd pas d'information.

   Correction : `double`, utilisant 11 bits pour encoder la position de sa virgule, contient seulement 53 bits pour le signe et les chiffres significatifs du nombre (la mantisse) alors qu'un `long` en utilise 64. Donc nécessairement, certains `long` ne sont pas représentables en `double` sans arrondi (concrètement : la conversion efface les 11 bits de poids faible et indique que la virgule se situe 11 chiffres à droite).

7. V Une interface peut contenir une `enum` membre.

   Correction : C'est autorisé par le langage. À noter que l'`enum` membre est alors statique.

8. V Le type des objets Java est déterminé à l'exécution.

   Correction : Impossible autrement : les objets n'existent pas avant.

9. V Tout seul, le fichier `A.java`, ci-dessous, compile :

```
public class A { final boolean a = 0; }
class B extends A { final boolean a = 1;}
```

   Correction : Si le doute portait sur le modificateur `final`, alors pas de problème car l'initialisation de `B` ne réaffecte pas une valeur à l'attribut `a` déclaré dans `A` : en effet, l'attribut déclaré dans `B` masque celui-ci.

10. F Une `enum` peut hériter d'une autre `enum`.

    Correction : Une `enum` est une classe héritant déjà de `Enum`, donc elle ne peut pas hériter d'une autre classe (de genre `enum` ou autre).

11. V Une `enum` peut avoir plusieurs supertypes directs.

    Correction : C'est en effet possible en implémentant une ou des interfaces.

12. F Quand, dans une classe, on définit une méthode de même nom qu'une méthode héritée, il y a nécessairement masquage ou redéfinition de cette dernière.

    Correction : Pas forcément : si la signature ne correspond pas, on est dans un cas de surcharge.

13. F Le type de l'argument de la méthode `add` d'une instance donnée de `LinkedList` est connu et interrogeable à l'exécution.

Correction : Ce type est seulement connu du compilateur, virtuellement associé à l'instance courante de `LinkedList`, mais oublié aussitôt la compilation terminée (*type erasure*).

14. V `HashSet<Integer>` est sous-type de `Set<Integer>`.

Correction : Oui : d'une part `HashSet` implémente `Set`, d'autre part, les deux types génériques sont ici paramétrés avec le même paramètre (`Integer`).

15. F `Deque<Integer>` est sous-type de `Deque<Object>`.

Correction : Non : le paramètre devrait être identique (invariance des génériques).

16. V Le compilateur autorise à déclarer une `enum` qui soit sous-type de `Iterable<Boolean>`.

Correction : Oui, les enums, comme les classes classiques, peuvent implémenter n'importe quelle interface.

17. V Une classe peut avoir plusieurs sous-classes directes.

Correction : Sans problème : l'héritage est contraint seulement dans l'autre direction (une seule superclasse directe pour une classe donnée).

18. F Une classe `final` peut contenir une méthode `abstract`.

Correction : Si c'était le cas, il serait alors impossible d'implémenter un jour cette méthode car on ne pourrait pas créer de sous-classe. Comme c'est absurde, c'est interdit par le compilateur.

19. V Une classe `abstract` peut contenir une méthode `final`.

Correction : Ici, pas de contradiction. Cela veut juste dire qu'une partie de l'implémentation ne sera pas modifiable par les sous-classes.

20. F Pour les types référence, sous-typage implique héritage.

Correction : Non, l'implémentation d'interface, par exemple, crée aussi du sous-typage.

21. F Dans la classe `B` ci-dessous, la méthode `f` de la classe `A` est masquée par la méthode `f` de `B` :

```
class A { private static void f() {} }
class B extends A { private static void f() {} }
```

Correction : On ne peut masquer que ce qui est hérité. Or la méthode `f` de `A` étant privée, n'est pas héritée.

22. V Il est interdit de placer à la fois `private` et `abstract` devant une déclaration de méthode.

Correction : En effet, `abstract` demande une redéfinition, or pour redéfinir, il faut hériter, mais les membres `private` ne sont pas héritables. Cette combinaison est donc absurde donc interdite par le compilateur.

23. F Tout seul, le fichier `Z.java`, ci-dessous, compile :

```
public class Z<T> {}
class W<Integer> extends Z<T> {}
```

Correction : Ici, tout est mélangé. À droite du nom de la classe qu'on déclare (`W`), on ne peut mettre qu'un paramètre qu'on introduit (`Integer` étant un nom de classe existant, il est peu probable qu'on ait voulu que ce soit le nom d'un paramètre) ; à droite du nom de la classe qu'on étend (`Z`), il faut remplacer le paramètre par un type bien défini dans le contexte (ça aurait pu être `Integer` : probablement l'intention du programmeur ici, mais certainement pas `T`, qui n'a pas d'existence en ce point du programme).

24. [V] Tout seul, le fichier `Z.java`, ci-dessous, compile (rappel : `Integer` est sous-classe de `Number`) :

```
public class Z<T extends Number> { static Z<Integer> w = new Z<>(); }
```

Correction : Aucun souci pour contrétiser `T` par `Integer`, car `Integer` est sous-type de la borne (`Number`).

25. [F] Tout seul, le fichier `Z.java`, ci-dessous, compile :

```
import java.util.function.*;
public class Z { Function<Object, Boolean> f = x -> { System.out.println(x); }; }
```

Correction : La lambda expression donnée ici ne peut pas implémenter la méthode `apply` de `Function`, car son type de retour est `void` (ou plutôt : elle ne retourne rien et, en tout cas, certainement pas `Boolean`).
Donc l'inférence vers `Function<Object, Boolean>` n'est pas possible.

26. [V] Tout seul, le fichier `Z.java`, ci-dessous, compile :

```
import java.util.function.*;
public class Z { Consumer<Object> f = System.out::println; }
```

Correction : La méthode `println` est de type de retour `void` et peut accepter des paramètres `Object`, elle peut donc servir à implémenter la méthode `accept` de `Consumer<Object>`. Donc l'inférence fonctionne ici.

27. [V] Tout seul, le fichier `LC.java`, ci-dessous, compile et garantit que toute instance de `LC` jamais créée (sauf modification de la classe `LC`) sera toujours soit une instance `LC.Cons`, soit de `LC.Empty`.

```
public class LC {
    private LC() {}
    public static class Empty extends LC { private Empty() {} }
    public static class Cons extends LC {
        public final int head; public final LC tail;
        public Cons(int head, LC tail) { this.head = head; this.tail = tail; }
    }
    public static Empty empty = new Empty();
}
```

Correction : En effet : le constructeur de `LC` est privé, donc il ne peut être appelé que depuis l'intérieur de `LC` et `LC` ne peut être étendue que depuis des classes imbriquées. Or à aucun endroit de la classe, `LC` n'est instanciée directement et les seules classes imbriquées sont `Empty` et `Cons` (et elles étendent `LC`).

28. [F] Même question que la précédente en remplaçant "instance" par "instance directe".

Correction : Là ça devient faux, car il est possible d'étendre `LC.Cons` depuis l'extérieur de `LC`. Il aurait fallu la marquer comme `final` ou rendre privé son constructeur.

29. [F] Il est possible, depuis l'extérieur, de créer une instance de `LC.Empty` différente de `LC.empty`.

Correction : Non car son constructeur est privé.

30. [V] Dans le programme ci-dessous, le type `Livre` est immuable :

```
public class Livre {
    public final String titre, auteur;
    private Livre(String auteur, String titre) { this.auteur = auteur; this.titre = titre; }
    public static final class Roman extends Livre {
        public Roman(String auteur, String titre) { super(auteur, titre); }
    }
    public static final class Essai extends Livre {
        public Essai(String auteur, String titre) { super(auteur, titre); }
    }
}
```

Correction : Ici `Livre` est une classe scellée correctement écrite (constructeur privé et sous-classes finales), dont les attributs sont `final` et eux-mêmes de type immuable (`String`). Donc toutes les instances de `Livre` sont garanties d'être non modifiables, donc le type `Livre` est immuable.