Nom, prénom : **Exemplaire n° :** 9

Contrôle 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.

Questions : *(Sauf mention contraire, les questions concernent Java 11.)*

1. F Le type d'une expression est calculé à l'exécution.

   Correction : Une expression est un morceau de code source.
   Ainsi, cette notion n'a de sens que pour le programmeur et pour le compilateur (qui, lui, calcule et vérifie le type des expressions : c'est le principe du typage statique), mais plus aucun sens à l'exécution.

2. F Quand, dans une méthode, on définit et initialise une nouvelle variable locale de type `int`, à l'exécution, sa valeur est stockée dans le tas.

   Correction : Les variables locales sont stockées en pile ce qui permet de récupérer la mémoire quand on sort de la méthode (décrémentation du pointeur de pile de la taille du *frame*).

3. F Tout seul, le fichier `LC.java`, ci-dessous, compile et le type `LC` est scellé. [1]

```java
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 : C'est faux, car il est possible d'étendre `LC.Cons` depuis un autre fichier,. Ainsi `LC` possède un sous-type non scellé, ce qui l'empêche d'être elle-même scellée. Il aurait fallu marquer cette classe comme `final` ou rendre privé son constructeur pour corriger cela.
   Sinon, remarquons que le constructeur de `LC` est privé„ ce qui implique que ses sous-types directs ne peuvent être que ses classes imbriquées. Si ses deux classes imbriquées avaient été `elles-mêmes scellées` `LC` aurait été scellée.

4. V Certaines vérifications de type ont lieu à l'exécution.

   Correction : Quelques vérifications à l'exécution :
   - l'évaluation de **checkcast** (instruction ajoutée au code-octet pour les *down-casting* d'objet) ;
   - l'évaluation d'**instanceof** ;
   - l'évaluation de `getClass()` ;
   - l'évaluation de **invokevirtual** (instruction ajoutée au code-octet pour appe-

1. Type scellé : type `T` dont la liste des sous-types est connue et fixée définitivement à la compilation de `T`.

ler une méthode d'instance afin que la JVM opère la liaison dynamique : choix d'une méthode en fonction du type de l'objet récepteur).

5. F Les opérations sur un attribut `volatile` sont atomiques[2] (par rapport à celui-ci).

   Correction : Tout ce que `volatile` peut contribuer à rendre atomique, ce sont les lectures et écritures simples sur les types de 64 bits (`long` et `float`). Les accès simples aux autres variables sont atomiques de toute façon.
   En revanche, les opérations en plusieurs étapes (par exemple une incrémentation) ne sont pas rendues atomiques par `volatile`.

6. F Placer le mot-clé `synchronized` devant toutes les méthodes d'une classe les rend atomiques[3] (par rapport aux attributs de cette classe).

   Correction : Cela les rendrait atomiques seulement si c'était les seules méthodes accédant à ces attributs. Donc c'est faux si les attributs ne sont pas privés.

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

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

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

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

   Rappel : `public interface Function<T, R> { R apply(T t); }`

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

9. F Les objets sont typiquement stockés dans la pile.

   Correction : Non, ils sont typiquement stockés dans le tas.
   On aurait pu vouloir dire « toujours » au lieu de « typiquement ». Mais ce ne serait pas tout à fait vrai : la JVM est autorisée à optimiser en stockant des objets en pile s'ils sont référencés seulement localement (sans *alias* externe). Cette optimisation ne provoque aucune différence fonctionelle (aucune différence de comportement visible, si ce n'est la vitesse d'exécution).

10. V Une méthode ne peut pas être à la fois `private` et `abstract`.

    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.

11. V Tout seul, le fichier `Z.java`, ci-dessous, compile :

2. Voir 3.
3. Opération/Méthode atomique par rapport à un ensemble de variables : opération/méthode dont les accès, à cet ensemble de variables, effectués par une exécution de celle-ci ne sont pas entrelaçables avec les accès, à des variables du même ensemble, effectués par d'autres opérations ou méthodes.

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

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

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

12. [V] Tout seul, le fichier `Z.java`, ci-dessous, compile (rappel : `Integer` étend `Number`) :

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

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

13. [V] Java dispose d'un système de typage statique.

Correction : La preuve : le compilateur refuse les programmes mal typés.

14. [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.

15. [F] La durée de vie d'un attribut statique est celle d'une instance donnée de la classe.

Correction : Non, la durée de vie d'un attribut statique est liée à la durée d'existence de la classe dans la mémoire de la JVM (typiquement : toute la durée de l'exécution du programme).

16. [V] `Object[]` est un supertype de `Integer[]`.

Correction : En effet, les tableaux de Java sont covariants.

17. [F] Le même programme sera exécuté environ 2 fois plus rapidement sur un ordinateur dont le CPU (microprocesseur) a 4 cœurs que sur un ordinateur dont le CPU en a 2 (toutes les autres caractéristiques du matériel restant identiques par ailleurs).

Correction : Cela n'est vrai que si le programme a été conçu pour distribuer son travail équitablement sur 4 *threads*.
Or cela n'est pas toujours possible (certains algorithmes, de par leur nature mathématique, ne peuvent pas être distribués sur plusieurs *threads*).

18. [V] Dans le programme ci-dessous, le type `Livre` est immuable[4] :

```java
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 {
```

4. Type immuable : type dont les instances directes ou indirectes, présentes ou futures sont non modifiables.

```
8         public Essai(String auteur, String titre) { super(auteur, titre); }
9     }
10 }
```

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.

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

```
1 public class Z<T> {}
2 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).

20. [F] Toute classe dispose d'un constructeur sans paramètre.

Correction : Toute classe dont au moins un constructeur est écrit par le programmeur ne dispose que des constructeurs écrits par le programmeur (dont peut-être un sans paramètre... ou pas).
Dans le cas contraire, le compilateur ajoute le constructeur dit « par défaut ». Celui-ci n'a pas de paramètre.

21. [V] Appeler la méthode `start` sur une instance de `Thread` démarre un nouveau *thread*.

22. [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.

23. [F] Les variables locales peuvent être des variables partagées (entre *threads*).

Correction : Les variables locales étant stockées en pile, et chaque *thread* possédant sa propre pile, les variables locales ne sont pas partagées.

24. [F] La JVM interprète du code source Java.

Correction : La JVM ne comprend que le code-octet JVM. Il peut l'interpréter ou bien le compiler à la volée (« JIT ») vers du code natif.
C'est, au contraire, le rôle du compilateur (javac) de comprendre le code source Java.

25. [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`).

26. 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).
Par ailleurs, les `enum` sont non-extensibles (autrement que par les éventuelles classes-singleton imbriquées générées par le compilateur pour les constantes de l'enum) car ce sont des types finis.

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

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

28. V L'instruction ci-dessous a pour effet d'afficher : `1 1`.[5]

```
1  Stream.of(1,2,3).peek(x -> System.out.print(x + " ")).limit(1).forEach(System.out::print)
```

Correction : L'affichage sera `1 1` (et non `1 2 3 1`) car les calculs demandés par les opérations intermédiaires ne sont exécutés que s'ils sont utiles pour obtenir le résultat demandé par l'opération terminale (évaluation paresseuse).
Le fait que `limit` ait été appelée fait que `forEach` ne demandera le traitement que du premier élément du *stream* (c'est à dire « `1` »). Donc le paramètre de *peek* ne sera exécuté que pour cet élément-là.

29. F Dans la classe `B` ci-dessous, la méthode `f` définie dans `B` masque la méthode `f` héritée :

```
1  class A { private static void f() {} }
2  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.

30. V Le mot-clé `volatile` devant un attribut empêche les accès en compétition à celui-ci.

5. La méthode `Stream<T> peek(Consumer<T> ope)` de `Stream<T>` est une opération intermédiaire retournant un *stream* identique à `this`, mais dont le traitement exécutera `ope` sur chaque élément du *stream*.
La méthode `Stream<T> limit(int n)` est une opération intermédiaire de `Stream<T>` retournant un *stream* identique à `this` mais tronqué aux `n` premiers éléments.
La méthode `void forEach(Consumer<T> ope)` est une opération terminale de `Stream<T>` qui exécute `ope` sur chaque élément du *stream* sur lequel elle a été appelée.