Nom, prénom :

Contrôle de Compléments en Programmation Orientée Objet n° 1
(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 11.

Questions :

1. [V] Quand, dans une méthode, on définit et initialise une nouvelle variable locale avec une instruction de la forme `Integer x = 12;`, alors la valeur 12 est stockée dans le tas.

   Correction : Les variables locales sont stockées en pile, mais en l'occurrence, le type `Integer` n'est pas primitif, donc la valeur stockée en pile est une référence vers un objet du tas contenant la valeur 12 dans un champ.

2. [V] Quand « `this` » apparaît dans une méthode, sa valeur est le récepteur de l'appel courant à celle-ci.

   Correction : NB : il y a aussi un autre usage de `this`, pour désigner un constructeur de la classe courante, lorsqu'appelé au début d'un autre constructeur de cette même classe. Dans ce cas, `this` n'est pas une expression, donc pas d'influence sur la réponse.

3. [V] Toute classe possède au moins un constructeur.

   Correction : Oui, au pire le compilateur ajoute le fameux « constructeur par défaut » (qui ne prend pas de paramètre, et initialise les champs à leur valeur par défaut, à savoir `0` ou `null`).

4. [F] `Number` est supertype de `double`.

   Correction : `Number` est un type référence (type de toutes les références) et `double` est un type primitif, or les types référence et les types primitifs sont deux hiérarchies de types disjointes.

5. [F] Quand un objet n'est plus utilisé, il faut demander à la JVM de libérer la mémoire qu'il occupe.

   Correction : Non, le ramasse-miettes détermine automatiquement quels objets ne sont plus référencés et peuvent donc être libérés (ce qu'il va donc faire périodiquement sans qu'on ait à le demander).

6. [F] La ligne 12 du programme ci-dessous affiche « `1` ».

```
1  class Truc {
2      static int v1 = 0; int v2 = 0;
3      public int getV1() { return v1; }
4      public int getV2() { return v2; }
5      public Truc() { v1++; v2++; }
6  }
7
```

```
8  public class Main {
9      public static void main(String args[]) {
10         System.out.println(new Truc().getV1());
11         System.out.println(new Truc().getV2());
12         System.out.println(new Truc().getV1());
13     }
14 }
```

7. [F] La ligne 11 du programme ci-dessus affiche « `3` ».

   Correction : `v1` et `v2` n'ont pas le même statut. `v1` est statique et donc n'existe qu'en un seul exemplaire, incrémenté à chaque instanciation de `Truc` (donc 3 fois avant la ligne 15) . Donc La ligne 12 affiche « `3` ».

`v2`, elle, est un attribut d'instance, donc un nouvel exemplaire existe pour chaque nouvelle instance de `Truc`, dont la valeur vaut `1` à la sortie du constructeur. Donc la ligne 11 affiche « `1`'.

8. F Une interface peut avoir des instances directes.

   Correction : Non, seules les classes (non abstraites) peuvent avoir des instances directes. Une interface n'a d'ailleurs pas de constructeur.
   Les instances des interfaces sont des objets, mais tout objet a pour type le plus précis une classe.

9. V Tout objet existant à l'exécution est instance de `Object`.

   Correction : `Object` est par définition le type de tous les objets... et de `null`.

10. V Le polymorphisme par sous-typage permet de réutiliser, dans un nouveau fichier `G.java` une méthode `f` définie dans le fichier `F.java` (sans recompilation de ce dernier) avec des paramètres effectifs dont le type n'avait pas encore été programmé quand `F.java` avait été compilé.

    Correction : Tout à fait, cela est vrai pour peu que les types des paramètres soient des sous-types de ceux déclarés dans la signature de `f`. Le compilateur l'accepte (c'est le principe-même d'un sous-type), et cela fonctionne en pratique car les méthodes (et autres membres) du supertype sont héritées par le sous-type, garantissant que les paramètres passés possèdent bien toutes les méthodes (et autres membres) utilisées dans `f`.

11. V Il est plus facile de prouver qu'un programme se comporte correctement quand ses classes *encapsulent* leurs données que quand elles ne le font pas.

    Correction : En effet, l'encapsulation permet d'assurer qu'un membre n'est utilisé que depuis l'intérieur de la classe, donc la preuve ne nécessite que de regarder ce qui se passe dans la classe à valider.

12. F Une classe implémentant une interface `I` doit 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.

13. V La méthode `somme` ci-dessous s'exécute toujours normalement (sans lever `IndexOutOfBoundsException`) :

```
1  import java.util.List; import java.util.ArrayList;
2  public class PaquetDEntiers {
3      private final List<Integer> contenu; private final int taille;
4      public PaquetDEntiers(ArrayList<Integer> contenu) {
5          this.contenu = new ArrayList<>(contenu);
6          this.taille = this.contenu.size();
7      }
8      public int somme() {
9          int s = 0; for (int i = 0; i < taille; i++) { s += contenu.get(i); } return s;
10     }
11 }
```

    Correction : `IndexOutOfBoundsException` aurait pu se produire si la méthode `get` avait été appelée sur `this.contenu` avec un paramètre d'indice invalide. Ici, tous les indices de `0` à `taille - 1` sont utilisés. Or `taille` est la taille de `this.contenu` juste après sa construction.
    Donc si `this.contenu` n'est pas susceptible de voir sa taille diminuer en dessous de `taille`, il n'y a pas de prolbème. Or `this.contenu` est une liste instanciée dans la classe et dont la référence n'est jamais partagée (notamment, elle est stockée dans un attribut privé). Donc cette exception ne peut pas se produire.

Remarque : la création de la liste `this.contenu` depuis le paramètre `contenu` correspond à une copie profonde, d'où le fait que ce programme fonctionne bien.
Autre remarque : évidemment, enregistrer la taille de la liste dans un attribut séparé est redondant et créée des risques d'incohérences. Le programme aurait très bien pu fonctionner (en *mono-thread*) sans la copie profonde si on avait utilisé `contenu.size()` au lieu de `taille`.

14. [V] Le patron de conception « adaptateur » consiste à écrire une classe implémentant une interface donnée, à l'aide d'une autre classe qui fournit les fonctionnalités de cette interface sans l'implémenter.

15. [V] `javac` prend en entrée un code source Java et produit, en sortie, du code-octet.

    Correction : En effet. Compiler le code source sous cette forme le rend exécutable sur toutes les machines physiques pour lesquelles une JVM est implémentée.

16. [F] Dans la classe suivante :

```java
public class A {
    private int d;
    public A(int d) { this.d = d; }
    public int getD() { return d; }
}
```

    Pour s'assurer que l'appel à `getD` sur une même instance de `A` retourne toujours la même valeur, il est nécessaire ajouter, dans le constructeur, une copie profonde du paramètre `d`.
    (On suppose que tout est exécuté sur le même thread.)

    Correction : Ce n'est pas nécessaire car ce paramètre est de type primitif. Ainsi l'affectation `this.d = d` est déjà une copie profonde (vu que `d` n'a pas de « profondeur »).
    La remarque sur les *threads* a été ajoutée parce qu'il est possible, en toute généralité, que le constructeur et `getD` soient exécutés sur des *threads* différents. Dans ce cas là, il n'y a pas de garantie que l'attribut soit déjà initialisé quand `getD` lit sa valeur.
    Pour « réparer » cette classe en mode *multi-thrlead*, il faudrait ajouter `volatile` devant la déclaration de `d`, voire, dans ce cas précis, `final` (qui donne la même garantie de synchronisation), vu que `d` ne pourra plus être modifié après son initialisation.

17. [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`).

18. [F] Le système de sous-typage de Java est structurel (si une interface `T` possède toutes méthodes d'une autre interface `U`, avec des signatures compatibles, alors `T` est sous-type de `U`).

    Correction : Non, le système de type de Java est, au contraire, nominal et déclaratif : deux types sont le même type si et seulement si ils ont le même nom ; un type est sous-type d'un autre s'il a été déclaré en tant que tel via la clause `extends` ou `implements` (ou si c'est de base dans le langage).

19. [V] La classe d'un objet donné est connue et interrogeable à l'exécution.

    Correction : Tout à fait, tout objet contient une référence vers sa classe. Cela permet notamment à `instanceof` et à la liaison dynamique de fonctionner.

20. [V] Si `A` et `B` sont des types référence, `A` est sous-type de `B` si et seulement si toutes les instances de `A` sont aussi des instances de `B`.

    Correction : Tout à fait : la relation de sous-typage coincide avec la relation d'inclusion d'ensembles (pour les ensembles qui sont des types).

21. [V] Le type `byte` est primitif.

Correction : Il fait partie de la liste fixe des 8 types primitifs (dont le nom commence par une minuscule).

22. [F] Le type `String` est primitif.

Correction : `String` est un type référence (commence par une majuscule, sous-type de `Object`). Sinon, un `String` est représenté en Java par un tableau de `char`, dont la taille dépend de la longueur de la chaîne. Les primitifs prenant exactement 32 bits (ou 64), `String` ne pouvait pas êre primitif.

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

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

Correction : Il y a incompatibilité de type entre les litéraux « `0` » et « `1` » (entiers) et le type déclaré pour les variables « `a` » (`boolean`), donc ce programme ne compile pas.
Cela dit, cela était une erreur dans l'énoncé. Les variables étaient supposées être déclarées `int`.
Si cela avait été le cas, la réponse aurait été [V], avec l'explication suivante :
« 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. »
Le point est donc donné à tout le monde pour que ne soient pénalisés ni les étudiants ayant répondu [V] en ayant compris les enjeux de la question mais en n'ayant pas vu le « piège » involontaire, ni les étudiants ayant répondu [F] en ayant vu le piège, ni ceux qui n'ont pas répondu pour cause de perplexité par rapport aux intentions de l'auteur de la question !

24. [F] Dans la classe `B` ci-dessous, la méthode `f` de la classe `A` est surchargée par la méthode `f` de `B` :

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

Correction : Il n'y a surcharge en un point donné du programme que si plusieurs définitions de méthode de même nom existent dans le contexte courant. Or la méthode `f` de `A` étant privée, n'est pas héritée dans le contexte de `B` (et est de toute façon inacessible en dehors de `A`).

25. [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).

26. [F] Pour les types référence, sous-typage implique héritage.

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

27. [F] La dernière version de Java est Java 12.

Correction : Faux, c'est Java 13, sortie de 17 septembre dernier (c'est dans le cours !)

28. [F] Java est un langage orienté objet à prototypes.

Correction : Non, Java est un LOO à classes : il faut créer des classes pour instancier des objets, et on ne peut pas juste créer un prototype que l'on clone.

29. [V] La durée de vie d'un attribut non statique est celle d'une instance donnée de la classe.

Correction : Oui, un attribut non statique est aussi appelé attribut d'instance, ce qui veut dire qu'il est un constituant d'une instance donnée (de la classe dont il est un attribut).

30. F Avec `x` et `y` de type `Object`, après exécution de l'instruction `x = y;`, la variable `x` représente désormais une copie de l'objet représenté par `y`.

Correction : Une affectation copie seulement ce qu'il y a directement dans la variable. Pour les types référence comme `Object`, la variable contient une adresse, qui est copiée. Ainsi, `y` contiendra une adresse pointant sur le même objet que `x`, qui n'a donc jamais été copié ici.