Nom, prénom :

Partiel 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 8.

1. [F] Quand, dans une méthode, on définit et initialise une nouvelle variable locale de type `int`, 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*).

2. [V] Quand "`this`" est une expression, elle s'évalue comme l'objet sur lequel la méthode courante a été appelée.

   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. [F] Toute classe dispose d'un constructeur par défaut, sans paramètre.

   Correction : Toute classe non munie d'un constructeur écrit par le programmeur se verra ajouter automatiquement le constructeur par défaut par le compilateur.
   Les autres classes n'ont pas ce constructeur par défaut, mais seulement ceux que le programmeur aura écrits.

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

   Correction : `Object` 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. [V] ou [F] Une variable locale est toujours déclarée à l'intérieur d'un bloc d'instructions.

   Correction : En fait, cela dépend de ce qu'on entend par variable locale.
   La définition sur Wikipédia : *"En programmation informatique, une variable locale est une variable qui ne peut être utilisée que dans la fonction ou le bloc où elle est définie."*.
   Selon cette définition, l'énoncé est un pléonasme (donc [V]).
   Mais si on se focalise sur l'usage de la variable : variable locale = variable utilisable seulement dans un bloc d'instruction donné, alors il faut inclure (en ce qui concerne Java) les cas suivants :
   - variable déclarée entre les parenthèses d'un `for` :
     En effet,

     ```java
     {
        // ici, i n'existe pas encore
        for (int i = 0; i < 10; i++) System.out.println(i);
        // ici, i n'existe plus
     }
     ```

     est équivalent à :

     ```java
     {
        // ici, i n'existe pas encore
        {  // <- bloc implicite
            int i = 0; // ici i existe et vaut 0
            for (; i < 10; i++) System.out.println(i);
            // ici i existe et vaut 10
     ```

```
7       }
8       // ici, i n'existe plus
9   }
```

La discussion peut porter sur la réalité de ce bloc qui n'existait pas dans le code source, bien que tout se passe comme si c'était le cas.
Remarque : si l'instruction paramètre de `for` est elle-même un bloc, alors `i` n'est pas locale à ce bloc : en effet, quand celui-ci est exécuté, elle garde la valeur qu'elle avait lors de l'exécution précédente du bloc, de plus la variable est testée et incrémentée en dehors du bloc (dans les parenthèses du `for` : `i<10; i++`).
Cela dit, l'instruction `for`, elle-même, est toujours dans un bloc d'instruction. Donc ce cas là ne contredit pas l'énoncé.

— paramètre formel d'une méthode : l'usage est identique à celui d'une variable locale dans le corps de la méthode (= bloc d'instructions) ; pourtant la déclaration est dans les parenthèes avant le corps.
Autre indice : Java interdit la déclaration d'une variable locale ayant le même nom qu'un des paramètres de la méthode, ce qui prouve que les 2 notions sont conceptuellement la même chose, du moins dans l'esprit des auteurs du langage.

En suivant cette interprétation, la réponse serait F .

En raison de l'ambiguïté possible de la définition de variable locale (et vu qu'aucune définition "officielle" n'est donnée dans le cours), cette question n'est pas prise en compte dans le total des points.

6. F On écrit `public` devant la déclaration d'une variable locale pour qu'elle soit visible depuis les autres classes.

Correction : Les variables locales sont... locales ! Ceci signifie qu'elles n'ont du sens que dans le bloc où elles sont déclarées. Les modificateurs de visibilité ne s'y appliquent donc pas.

7. F Dès lors qu'un objet n'est plus utilisé, il faut penser à demander à Java 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).

8. F La ligne 11 du programme ci-dessous affiche "2".

```
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 }
```

9. F La ligne 12 du programme ci-dessus affiche "1".

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".

10. 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).

11. F La plateforme Java est adaptée à la programmation système.

Correction : Les méthodes d'accès au système fournies par l'API Java sont suffisament abstraites pour être communes à toutes les plateformes pour lesquelles il existe une implémentation de la JVM. Elles ne donnent donc pas une vision assez bas niveau pour manipuler les primitives d'un système d'exploitation donné.

12. V Dans un fichier source Java, une instruction se situe nécessairement (pas forcément seule) entre une accolade ouvrante et une accolade fermante.

Correction : C'est principalement une question de définition. Selon la JLS, les instructions (*statements*) sont des éléments syntaxiques qu'on peut placer dans un bloc (qui lui-même est délimité par des accolades et peut aussi servir d'instruction) : `https://docs.oracle.com/javase/specs/jls/se8/html/jls-14.html#jls-Statement`.
Remarque : il y a d'autres paires d'accolades que les blocs, comme par exemple les corps de classe ou d'interface.

13. 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.

14. 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.

15. V Tout objet existant a l'exécution est instance de `Object`.

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

16. V Le polymorphisme par sous-typage permet de réutiliser une méthode `f(...)`, avec des paramètres effectifs de types différents entre deux utilisations, sans recompiler `f(...)`.

Correction : Tout à fait :
- Pour les valeurs primitive, l'appel peut être précédé d'une conversion comme **i2f** (on parle de la compilation de l'appel, pas celui de la méthode, qui a été compilée séparément). Le sous-typage assure que dans tous les cas de figure, une telle conversion est soit existante, soit non nécessaire.
- Pour les références d'objet, l'appel se fait sans précaution particulière. Les règles d'implémentation d'interface et d'héritage assurent que tous les membres de l'objet accédés dans la méthode appelée sont bien définis pour l'objet passé en paramètre.

17. 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.

18. 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).

19. F Une classe implémentant une interface doit implémenter/redéfinir toutes les méthodes déclarées dans l'interface.

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.

20. V Pour faire un *downcasting*, on doit demander explicitement le transtypage.

Correction : Le *downcasting* est une opération périlleuse (primitifs : perte d'information ; références : exception si à l'exécution l'objet référencé n'a pas le type demandé), il doit donc être demandé explicitement.

21. F La méthode `somme` ci-dessous s'exécute toujours sans erreur (exception) :

```
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          if (contenu != null) this.contenu = contenu;
6          else this.contenu = Collections.emptyList(); // initialisation à liste vide
7          this.taille = this.contenu.size();
8      }
9      public int somme() {
10         int s = 0; for (int i = 0; i < taille; i++) { s += contenu.get(i); } return s;
11     }
12 }
```

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 *alias*ée, 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.

22. V Java dispose d'un système de typage statique.

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

23. F Le code source doit être compilé en code-octet avant chaque exécution.

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

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

Correction : On aurait pu vouloir dire “toujours” au lieu de “typiquement”. Mais ce n’est 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).

25. 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 *downcasting* d’objet) ;
- l’évaluation d’**instanceof** ;
- l’évaluation de `getClass()` ;
- l’évaluation de **invokevirtual** (instruction ajoutée au code-octet pour appeler une méthode d’instance afin que la JVM opère la liaison dynamique : choix d’une méthode en fonction du type de l’objet récepteur).

26. 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`).

27. F Un transtypage de référence se traduit toujours par une instruction spécifique dans le code-octet.

Correction : Seulement pour le *downcasting*,. En effet, selon le principe du “ça passe ou ça casse”, l’instruction à ajouter ne sert qu’à vérifier le type de l’objet ; or pour l’*upcasting*, la vérification statique du type assure que l’objet est toujours du bon type, donc la vérification à l’exécution est inutile, donc aucune instruction dans le code octet n’est nécessaire.

28. F On peut déclarer une classe non imbriquée avec la visibilité **private**.

Correction : Seuls **public** et *package-private* sont autorisés. En Java, **private** veut dire “privé pour la classe”, ce qui n’a pas de sens pour une classe non imbriquée.
Cela dit, on aurait pu donner un sens différent à **private** en dehors de la classe, comme par exemple “privé pour le fichier” (c’est le choix du langage Kotlin, par exemple).

29. V Dans une classe `B`, membre statique de `A` (on suppose que `B` ne contient pas elle-même de définition de type imbriqué), **this** désigne toujours une instance de `B`.

Correction : **this** désigne toujours une instance de la classe “la plus proche” (dans l’ordre d’imbrication).

30. V La classe d’un objet donné est connue et interrogeable à l’exécution.

Correction : Cf. la correction de la question 25.
Ces différents cas de figure fonctionnent car tout objet contient une référence vers l’objet-classe de sa classe.

31. F La conversion de **int** vers **float** ne perd pas d’information.

Correction : **float**, utilisant 8 bits pour encoder la position de sa virgule, contient seulement 24 bits pour les chiffres significatifs du nombre (la mantisse) alors qu’un **int** en utilise 32. Donc nécessairement, certains **int** ne sont pas représentables en **float** sans arrondi (concrètement : la conversion efface les 8 bits de poids faible et indique que la virgule se situe 8 chiffres à droite).

32. V Une interface définit un sous-type de `Object`.

Correction : Oui, une interface définit un type référence, donc sous-type de `Object`. Cela est cohérent car toutes les instances d'une interface sont des objets (instances des classes implémentant l'interface).

33. [V] Une interface peut avoir une classe membre.

Correction : C'est autorisé par le langage. À noter que la classe membre est alors statique.
Remarque : cela permet, indirectement, de mettre des membres privés dans une interface (alors que c'est interdit de le faire directement).

34. [F] Java est plus ancien que C++.

Correction : C++ : années 80, Java : années 90.
Cela dit j'ai réalisé que l'âge de C++ n'était pas dans le cours. Donc cette question est retirée du décompte des points.

35. [F] Le type d'une expression est calculé à l'exécution.

Correction : La notion d'expression n'a pas de sens à l'exécution : c'est un morceau de texte du programme qui n'existe plus dès que le code est compilé.
Il se trouve que l'expression est l'unité de base pour la vérification de type statique.

36. [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).

37. [V] Le type des objets Java est déterminé à l'exécution.

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

38. [V] Le type `char` est primitif.

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

39. [F] Le type `Object` est primitif.

Correction : Les objets sont considérés comme non primitifs (composites, en toute généralité).

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

Correction : La JVM interprète seulement (ou bien compile à la volée, cf. JIT) du code-octet, mais en aucun cas du code source. C'est le rôle du compilateur (javac) de comprendre le code source.