Questionnaire à choix multiples

Classes, objets, héritage

I Notions de base en POO et Java

1. On considère le code suivant...

```java
// dans le fichier A.java
public class A {
    public int nbr;
}

// dans le fichier Test.java
public class Test {
    public static void main(String [] args){
        A a;
        a.nbr = 5;
    }
}
```

Quelles sont les affirmations correctes (plusieurs réponses possibles) ?

☐ a. Ce programme ne compile pas car il n'y a pas de constructeur ni aucune méthode dans la classe A !

☐ b. Ce programme ne compile pas car dans le `main()` l'objet `a` n'a pas été initialisée.

☐ c. Ce programme compile mais, lorsqu'on l'exécute, la machine virtuelle lève une `NullPointerException` à la première ligne du `main()` et le programme s'arrête car cette exception n'est pas récupérée.

☐ d. Ce programme compile mais, lorsqu'on l'exécute, la machine virtuelle lève une `NullPointerException` à la deuxième ligne du `main()` et le programme s'arrête car cette exception n'est pas récupérée.

2. On considère le code suivant...

```java
// dans le fichier A.java
public class A {
    private int nbr;

    public void incrementerNbr() {
        nbr ++ ;
    }

    @Override
    public String toString() {
        return "nbr = " + nbr ;
    }
}
```

```java
// dans le fichier Test.java
public class Test {
    public static void main(String [] args){
        A a1 = new A();
        A a2 = new A();

        a1.incrementerNbr();
        a2 = a1;
        a2.incrementerNbr();

        System.out.println("a1 : " + a1.toString());
        System.out.println("a2 : " + a2);
    }
}
```

Quelles sont les affirmations correctes (plusieurs réponses possibles) ?

☐ a. Ce programme ne compile pas car lorsqu'on crée les objets dans le main, on utilise un constructeur sans paramètre et qu'il n'y a pas de constructeur sans paramètre dans la classe A !

☐ b. Ce programme ne compile pas car, dans le second `println()` du `main()`, la concaténation d'une chaine et d'une référence (`"a2 : "+ a2`) ne veut rien dire !

☐ c. Ce programme compile et, lorsqu'on l'exécute, il s'affiche :

```
a1 : nbr = 1
a2 : nbr = 2
```

☐ d. Ce programme compile et, lorsqu'on l'exécute, il s'affiche :

```
a1 : nbr = 2
a2 : nbr = 2
```

☐ e. Ce programme compile et, lorsqu'on l'exécute, il y a une fuite mémoire. En effet, lorsqu'on exécute `a2 = a1 ;` dans le `main()`, il n'y a plus aucune référence qui manipule l'objet initialement créé à la 2ème ligne du `main()`. Cet objet ne peut plus être accédé. Il reste, tout seul, très triste, "perdu" en mémoire, jusqu'à la fin du programme... C'est une fuite mémoire.

3. On considère le code suivant...

```java
// dans le fichier A.java
public class A {
    private int nbr;
    public A(int nbr) {
        this.nbr = nbr;
    }

    @Override
    public String toString() {
        return "nbr = " + nbr ;
    }
```

```
}

// dans le fichier Test.java
public class Test {
    public static void main(String [] args){
         A a = new A();
        System.out.println("a : " + a);
    }
}
```

Quelles sont les affirmations correctes (plusieurs réponses possibles) ?

☐ a. Ce programme ne compile pas car il n'y a pas de constructeur sans paramètre dans la classe A !

☐ b. Ce programme compile et, lorsqu'on l'exécute, il s'affiche :

```
a : nbr = 0
```

☐ c. Ce programme compile et, lorsqu'on l'exécute, il s'affiche :

```
a : nbr =   <<<< et ici un nombre aléatoire >>>>
```

4. On considère le code suivant...

```
// dans le fichier Moteur.java
public class Moteur {
    private int puissance;
    public Moteur(int puissance) {
        this.puissance = puissance;
    }

    /** Constructeur par défaut. Puissance par défaut : 1000 */
    public Moteur() {
        this(1000);
    }

    public Moteur(Moteur other) {
        this.puissance = other.puissance;
    }

    public void setPuissance(int puisance) {
        this.puissance = puissance;
    }

    @Override
    public String toString() {
        return "puissance= " + puisance;
    }
}

// dans le fichier Avion.java
```

```
public class Avion {
    private int nbPassagers;
    private Moteur m;

    public Avion(int nbrP, Moteur m) {
        this.nbPassagers = nbPassagers;
        this.m = m;
    }

    public Moteur getMoteur() {
        return m;
    }

    public int getNbPassagers() {
        return nbrPassagers;
    }

    @Override
    public String toString() {
        return "nbr de passagers= " + nbPassagers + " moteur " +
           moteur ;
    }

    public static void main(String [] args){
        Moteur m = new Moteur();
        Avion a = new Avion(12, m);
        m.setPuissance(1);

        System.out.println("a, début : " + a);

        Moteur m2 = a.getMoteur();
        m2.setPuissance(0);

        int n = a.getNbPassagers();
        n = 0;

        System.out.println("a, fin : " + a);
    }
}
```

Quelles sont les affirmations correctes (plusieurs réponses possibles) ?

☐ a. Ce programme affiche

```
a, debut : nbr de passagers= 12  moteur puissance= 1000
a, fin : nbr de passagers= 12  moteur puissance= 1000
```

☐ b. Ce programme affiche

```
a, debut : nbr de passagers= 12  moteur puissance= 1
a, fin : nbr de passagers= 12  moteur puissance= 1000
```

☐ c. Ce programme affiche

```
a, debut : nbr de passagers= 12  moteur puissance= 1
a, fin : nbr de passagers= 12  moteur puissance= 0
```

☐ d. Ce programme affiche

```
a, debut : nbr de passagers= 12  moteur puissance= 1
a, fin : nbr de passagers= 0  moteur puissance= 0
```

☐ e. Si on voulait éviter que le moteur de l'avion soit modifié quand on modifie `m` à la 3ème ligne de la méthode `main()`, on pourrait réécrire le constructeur d'`Avion` ainsi :

```
public Avion(int nbrP, Moteur m) {
    this.nbPassagers = nbPassagers;
    // on fait une copie du moteur passé en paramètre.
    // Comme cela, l'avion possède son propre objet moteur
    this.m = new Moteur(m);
 }
```

☐ f. Si on voulait éviter que le moteur de l'avion puisse être modifié lorsqu'on y accède au moyen de l'accesseur `getMoteur()`, on pourrait réécrire cet accesseur ainsi :

```
public Moteur getMoteur() {
    // on retourne une copie du moteur
    // Comme cela, si quelqu'un modifie la copie retournée,
    // le moteur de this n'est pas modifié
    return new Moteur(m);
}
```

5. On rappelle que `IllegalArgumentException` dérive de la classe `RuntimeException` et on considère le code suivant...

```
// dans le fichier A.java

/** INVARIANT DE CLASSE pour la classe A :  nbr est toujours > 0 */
public class A {
    int nbr;

    public void setNbr(int nbr) {
        if(nbr <= 0 {
            throw new IllegalArgumentException("nbr doit etre
                strictement positif");
        }
        this.nbr = nbr;
    }

    public void getNbr() {
        return nbr ;
    }

    public static void main(String [] args){
        A a = new A();
        a.setNbr(-5);
    }
}
```

Quelles sont les affirmations correctes (plusieurs réponses possibles) ?

☐ a. Ce programme ne compile pas, car le prototype de la méthode `setNbr(int nbr)` ne déclare pas qu'il peut lever une exception de type `IllegalArgumentException`. La méthode devrait être écrite ainsi :

```
public void setNbr(int nbr) throws IllegalArgumentException {
    etc
}
```

☐ b. Ce programme ne compile pas car la méthode `setNbr(int nbr)` peut lever une exception et que, dans la méthode main, son appel n'est pas placé dans un bloc `try / catch`.

☐ c. Ce programme compile sans erreur. A l'exécution, une exception est levée. Le programme s'arrête car cette exception n'est jamais récupérée.

☐ d. Ce programme compile sans erreur. Toutefois, compte tenu de l'invariant de classe souhaité pour les instances de la classe `A` (en commentaire au dessus de la classe) il y a une grosse erreur de conception dans la déclaration de l'attribut `int nbr`.

☐ e. Ce programme compile sans erreur... Toutefois, compte tenu de l'invariant de classe souhaité pour les instances de la classe `A`, il y a une grosse erreur de conception : la classe `A` devrait être munie d'au moins un constructeur ! Sans quoi, à la construction de l'objet, l'invariant de classe n'est pas respécté.

☐ f. La classe `A` est vraiment trop bête et ne pourra pas servir à grand chose. Quand même, les profs pourraient faire des exemples moins absurdes pour un QCM...

6. Quelles sont les affirmations correctes (plusieurs réponses possibles) ?

☐ a. Une variable de type `double` est codée en Java sur 8 octets, avec la norme IEEE754.

☐ b. Une variable de type `int` est codée en Java sur 2 octets. Sa valeur min est -2^{15}. Sa valeur max est $2^{15} - 1$.

☐ c. Les types primitifs Java sont : `boolean`, `char` (un caractère), `byte` (1 entier sur 1 octet), `short`, `int`, `long`, `float`, `double`.

☐ d. En Java, la valeur d'une référence est l'adresse dans la mémoire de machine virtuelle Java de l'objet référencé (ou `null` si la référence ne référence aucun objet).

7. On considère le code suivant...

```
public class A {
    private double a;

        public A() {
        a = 0;
    }
    public void setA(double a) {
        this.a = a;
    }
    public double getA() {
        return a;
```

```
    }
    // et ici le reste de la classe A
}

public class Test {
        public static void main(String [] args){
            int [ ] tab1 = new int [12];
            A   [ ] tab2 = new A[12] ;

            tab1 [5] = 9;
            tab2[5].setA(123);
            System.out.println("a de tab[5] vaut " + tab2[ 5 ].getA())
                ;
     }
}
```

Quelles sont les affirmations correctes (plusieurs réponses possibles) ?

☐ a. Ce programme ne compile pas, car dans l'appel de `setA(double)` on passe un entier (123), alors que la méthode attend un `double`.

☐ b. Ce programme compile, mais à l'exécution une `NullPointerException` est levée lors de l'exécution de `tab2[5].setA(123)`. Le programme s'arrête et la dernière ligne `System.out.println(...)` n'est pas exécutée.

8. On reprend la classe `Rational` étudiée lors de la première séance et considère le programme principal suivant :

```
Rational a, b;
a = new Rational(2,3);
b = a;
a.add(b); // Addition en place
a = null;
```

Que vaut `b` à la fin du code suivant ?

☐ a. $\frac{2}{3}$

☐ b. $\frac{4}{3}$

☐ c. 0.

☐ d. `null`.

☐ e. Aucune proposition ci-dessus n'est correcte.

9. Un attribut statique (`static`)...

☐ a. ...ne peut être utilisé que par une méthode statique.

☐ b. ...est un attribut relatif à une classe.

☐ c. ...est un attribut partagé par toutes les instances de la classe.

☐ d. ...est un attribut dont la valeur ne change pas (une constante).

☐ e. ...est un attribut inaccessible en-dehors de la classe.

☐ f. Aucune proposition ci-dessus n'est correcte.

10. On considère le code suivant...

```
public class A {
    private double a;
    float b;
    public int c;

    private void methodeA() {
        ...
    }
    void methodeB() {
        ...
    }
    public void methodeV() {
        ...
    }
}
```

Quelles sont les affirmations correctes (plusieurs réponses possibles) ?

☐ a. Dans le code d'une autre classe que A, il n'est jamais possible d'accéder à l'attribut `a`, et à la méthode `methodeA()` d'un objet de type `A`.

☐ b. Dans le code d'une autre classe que A, il n'est jamais possible d'accéder à l'attribut `b`, et à la méthode `methodeB()` d'un objet de type `C`.

☐ c. Dans le code d'une autre classe que A, située dans le même paquetage que A, il est possible d'accéder à l'attribut `b`, et à la méthode `methodeB()` d'un objet de type `A`.

☐ d. Dans le code d'une autre classe que A, quel que soit son paquetage, il est possible d'accéder à l'attribut `c`, et à la méthode `methodeC()` d'un objet de type `A`.

☐ e. On appelle "API (application programming interface) de A" l'ensemble des champs `c` et `methodeC()` de la classe A.

☐ f. On appelle "API (application programming interface) de A" l'ensemble des champs `b`, `c`, `methodeB()`, `methodeC()` de la classe A.

☐ g. La visibilité des champs `b` et `methodeB()` de la classe A est appelée "visibilité par défaut" dans la classe A.

☐ h. En accord avec le "principe d'encapsulation en POO", une bonne pratique (un peu naïve, mais plutôt bonne !) consiste à déclarer "sans réfléchir" les attributs d'une classe `private`. Ce n'est que quand on a une bonne raison qu'on choisit un autre qualificateur de visibilité pour un attribut.

11. Gestion de la mémoire en Java :

☐ a. On peut détruire explicitement un objet de la mémoire avec la méthode `finalize()`.

☐ b. Il ne peut y avoir qu'une seule référence à la fois sur un objet.

☐ c. Le ramasse-miettes (*garbage collector*) s'occupe pour nous de la désallocation de la mémoire.

☐ d. Même si son exécution est automatique, on peut forcer le lancement du ramasse-miettes si on le désire.

12. Dans une classe, on considère la méthode suivante :

```
public static void methode(int a, String s, String[] tab) {
    a = 1;
    s = "Après";
    String[] tab2 = tab;
    tab2[0] = "Après";
}
```

Supposons que l'on réalise l'appel `methode(aPrime, sPrime, tabPrime)`, où les variables ont été initialisées par : `int aPrime = 0; String sPrime = "Avant"; String [] tabPrime = { "Avant"}` avant l'appel. Que valent ces variables après l'appel ?

☐ a. `0, "Avant", {"Avant"}`

☐ b. `0, "Avant", {"Après"}`

☐ c. `0, "Après", {"Avant"}`

☐ d. `0, "Après", {"Après"}`

☐ e. `1, "Avant", {"Avant"}`

☐ f. `1, "Avant", {"Après"}`

☐ g. `1, "Après", {"Avant"}`

☐ h. `1, "Après", {"Après"}`

13. Association entre classes. On considère les classes suivantes :

```
class X { ... }
class Y { // Version 1
    private X x;
    Y(X x) {
        this.x = x;
    }
}
class Y { // Version 2
    private X x;
    Y(X x) {
        this.x = new X(x);
    }
}
```

☐ a. Pour les versions 1 et 2, l'attribut `x` pourra permettre de déléguer une partie des calculs.

☐ b. En version 1, l'état de l'attribut `x` peut changer, alors qu'aucun mutateur n'est défini dans la classe `Y`.

☐ c. En version 2, quand le ramasse-miettes peut détruire un objet de classe `Y`, il peut aussi détruire son attribut `x`.

☐ d. Aucune proposition ci-dessus n'est correcte.