IFT1025 - Programmation 2 - Examen Intra

Professeur : Nicolas Hurtubise

Mardi 20 février 2018, 15h30 - 17h20

Prénom, ***Nom*** : _____, _____

Matricule : _____ ***Programme d'études*** : _____

0 Directives pédagogiques

- **Vous pouvez détacher les deux dernières pages (les Annexes A, B et C) du questionnaire**
- À la fin de l'examen, assurez-vous de remettre **toutes** les feuilles (questionnaire + feuille de réponses)

- L'examen est d'une durée de *1h50*
- Aucune documentation permise, à l'exception de deux feuilles de notes au format $8\frac{1}{2}$x11 recto-verso, **écrites de votre main**. Aucun appareil électronique n'est permis.

- Sur la feuille de réponses, dans la boite en haut à gauche, indiquez le sigle du cours (IFT1025), la faculté (FAS), la date, et votre signature.
- Sur la feuille de réponses, remplissez la section a. (NOM et PRÉNOM), la section b. (MATRICULE), et la section c. (VERSION 0). Noircir les ovales appropriés. Laissez la section d. vide.
- Assurez-vous d'indiquer vos *Prénom, Nom* et *Matricule* sur **chaque page du questionnaire**. Profitez-en pour vous assurer que vous avez bien un total de **13 pages** (dont deux pages d'annexes à la fin)
- **Tout plagiat entraînera la note de 0% automatiquement**

- Répondez aux questions 1.1 à 1.4 sur la **feuille de réponses** en noircissant un des ovales par question
- Répondez aux autres questions directement sur le **questionnaire**

1 Questions diverses (30%)

Barème pour les questions 1.1 à 1.4 :

- +1 pour une bonne réponse
- -0.5 pour une mauvaise réponse
- 0 pour pas de réponse

Barème pour la question 1.5 :

- +2 pour un affichage correct ou Erreur si l'expression cause une erreur
- 0 sinon

La proportion de points obtenus sur le maximum atteignable pour cette section sera reportée sur 30%.
Le score minimum est de 0 point (la correction négative n'influence pas les autres sections de l'examen).

Soient les définitions de classes dans l'Annexe A

1.1 Indiquez si les déclarations suivantes sont valides ou invalides

```
C c = new C();

 1. Whooperable w1 = new A();              // a. Valide    b. Invalide
 2. A a = new A();                         // a. Valide    b. Invalide

 3. B b1 = new B();                        // a. Valide    b. Invalide
 4. B b2 = c;                              // a. Valide    b. Invalide

 5. Whooperable w2 = new Whooperable();    // a. Valide    b. Invalide
 6. Whooperable w3 = c;                    // a. Valide    b. Invalide
 7. Whooperable w4 = new B();              // a. Valide    b. Invalide
```

1.2 Indiquez si les *casts* suivants sont valides ou invalides :

```
C c = new C();

 8. (A) c                                  // a. Valide    b. Invalide
 9. (B) c                                  // a. Valide    b. Invalide
10. (Whooperable) c                        // a. Valide    b. Invalide
11. (A) ((B) c)                            // a. Valide    b. Invalide
12. (A) ((Whooperable) ((C) ((B) c)))      // a. Valide    b. Invalide
```

1.3 Indiquez s'il serait valide ou invalide de faire les choses suivantes :

```
Comme pour les questions précédentes :
a = Valide    b = Invalide

13. Ajouter dans la classe C une nouvelle méthode :
    public void foo(int x) { /* Code ici ... */ }

14. Ajouter dans la classe C une nouvelle méthode :
    public static void qux() {
        System.out.println("Static !");
    }

15. Ajouter dans la classe C une nouvelle méthode :
    public static void twado() {
        System.out.println(this.bar());
    }


16. Écrire une sous-classe :
    public abstract E extends B { /* Code ici ... */ }
```

17. Écrire une sous-classe :
```
public F extends C { /* Code ici ... */ }
```

18. Écrire une classe :
```
public AAAAAAAAAAA extends A {
    /* pas de code dans la classe */
}
```

19. Écrire une classe :
```
public Jimmy implements Whooperable {
    /* pas de code dans la classe */
}
```

20. Ajouter dans la classe B une nouvelle méthode :
```
public String getA() {
    return this.a;
}
```

21. Ajouter dans la classe B une nouvelle méthode :
```
public String getA() {
    return super.a;
}
```

22. Ajouter dans la classe C une nouvelle méthode :
```
public String getB() {
    return super.b;
}
```

1.4 Indiquez si les expressions suivantes évaluent à `true`, `false` ou causent une `erreur`

```
B b = new B(); C c = new C();
```

```
23. b instanceof A              // a. true      b. false      c. erreur
24. c instanceof A              // a. true      b. false      c. erreur
25. b instanceof C              // a. true      b. false      c. erreur
26. b instanceof Object         // a. true      b. false      c. erreur
27. ((C) b) instanceof C        // a. true      b. false      c. erreur
28. c.bar() instanceof String   // a. true      b. false      c. erreur
```

1.5 Écrivez tout ce qui est affiché lors de son évaluation en dessous (considérez tous les `System.out.println()` impliqués par l'expression)

Si l'expression cause une erreur, écrivez seulement "Erreur"

```
// Soient les déclarations suivantes :
B b1 = new B(), b2 = new C();
C c = new C();
Whooperable w = c;
```

```
b1.foo();                                   |   c.foo();
                                            |
Affiche :                                   |   Affiche :
                                            |
                                            |
                                            |
                                            |
b2.baz();                                   |   w.whoop();
                                            |
Affiche :                                   |   Affiche :
                                            |
                                            |
                                            |
                                            |
w.foo();                                    |   b2.whoop();
                                            |
Affiche :                                   |   Affiche :
                                            |
                                            |
                                            |
                                            |
                                            |
((B) w).foo();                              |   ((Whooperable) b2).whoop();
                                            |
Affiche :                                   |   Affiche :
                                            |
                                            |
                                            |
                                            |
                                            |
```

2 Tableaux et références (10%)

Soit la classe `Foo` disponible dans l'Annexe B

Écrivez ce qu'affiche chaque `System.out.println`. Écrivez seulement "Erreur" si vous croyez que l'opération cause une erreur à la compilation ou à l'exécution.

```
// --- Fichier Question1.java ---
public class Question1 {

    public static int somme(float a, float b) {
        float c = a + b;

        return (int) c;
    }

    public static int max(int[] nums) {

        int max = 0;

        for(int i=0; i<nums.length; i++) {
```

```
            max = Math.max(nums[i], max);
        }

        return max;
    }

    public static void main(String[] args) {

        float a = 5.5f;
        float b = 6.6f;

        System.out.println(somme(b, a));
        // *** Affiche :

        System.out.println(somme(0.2f, 0.2f) + "" + somme(0.2f, 0.2f));
        // *** Affiche :

        int[][] tabs = {{6,6,6},{6,6,6}};
        int[] t1 = tabs[1];
        int[] t2 = {6,6,6};

        System.out.println(tabs[0] == tabs[0]);
        // *** Affiche :

        System.out.println(t1 == t2);
        // *** Affiche :

        System.out.println(tabs[0] == t1);
        // *** Affiche :

        System.out.println(tabs[1] == t1);
        // *** Affiche :

        System.out.println(max(tabs[1]));
        // *** Affiche :

        Foo fooTab = new Foo(t1);

        System.out.println(fooTab.egal(t1));
        // *** Affiche :

        System.out.println(fooTab.egal(t2));
        // *** Affiche :

        System.out.println(fooTab.egal(fooTab));
        // *** Affiche :
    }
}
```

3 Programmation Orientée Objet (25%)

Un `Livre` possède :

- Un titre (texte)
- Un nombre de pages (entier)
- Un prix régulier (entier - prix en nombre de cents)
- Un pourcentage de rabais (nombre entier compris entre 0 et 100)

Avec une instance de `Livre`, on peut :

- Calculer le prix de vente (entier, en nombre de cents) avec la formule : `prix régulier * (100 - rabais)/100`
- Utiliser `System.out.println(monSuperLivre)` pour afficher la description du livre dans le format suivant :

```
Titre - Nombre de pages - Prix de vente dans le format 0.00$
```

Écrivez une classe *abstraite* qui représente un livre.

Le *constructeur* d'un `Livre` doit permettre de définir le titre, le nombre de pages, le prix régulier et le pourcentage de rabais.

Choisissez des noms clairs pour vos méthodes et attributs. Assurez-vous de respecter le principe d'encapsulation. Écrivez des `getters/setters` pour lire/modifier les attributs. Commentez votre code lorsque nécessaire (il n'est pas obligatoire d'utiliser la `JavaDoc`, pas nécessaire de commenter vos `getters/setters`).

4 Héritage & Structures de données

4.1 Code (30%)

Un `Dictionnaire` est un type particulier de livre.

En plus de posséder les attributs et méthodes d'un livre, un dictionnaire contient une liste de mots associés à une définition.

La liste d'associations `mot => définition` sera stockée dans un attribut :

```
private ArrayList<Paire<String, String>> definitions;
```

La définition de la classe `Paire` et un rappel sur l'utilisation de la classe `ArrayList` sont disponibles dans l'*Annexe C*.

Les méthodes suivantes doivent être définies pour manipuler les définitions de mots :

```
public String getDefinition(String mot)

public void setDefinition(String mot, String definition)
```

La méthode `getDefinition(String mot)` permet de récupérer la définition d'un mot dans le dictionnaire. Si le mot n'est pas compris dans le dictionnaire, la chaîne vide `""` est retournée.

La méthode `setDefinition(String mot, String definition)` permet d'ajouter ou de mettre à jour la définition d'un mot. Si le mot n'est pas déjà présent dans le dictionnaire, on l'ajoute avec sa définition, sinon, on met à jour la définition du mot.

Notez que puisqu'il s'agit d'un `Dictionnaire` à petit budget, les mots du `Dictionnaire` ne sont pas nécessairement dans l'ordre alphabéthique. Autrement dit, si le mot "banane" a été ajouté après le mot "rouge", la définition de "banane" arrivera après dans l'ordre.

On peut demander le $N^{ième}$ mot du dictionnaire en utilisant les méthodes :

```
public String getDefinition(int index)
```

On peut afficher la description d'un dictionnaire en utilisant `System.out.println(monSuperDictionnaire)`, ce qui affiche dans le format :

```
...affichage habituel d'un Livre... - Nombre de définitions
```

En affichant la description d'un livre comme d'habitude, mais en ajoutant le nombre de définitions à la fin.

Notez que si on modifie le code pour afficher un `Livre`, ça devrait également modifier la façon dont on affiche un `Dictionnaire`

Écrivez une classe `Dictionnaire` qui est une sous-classe de `Livre`. N'oubliez pas d'instancier votre `ArrayList<...> definitions` dans le constructeur de `Dictionnaire`.

Notez : si vous voulez vous faciliter la tâche, écrivez plus de méthodes que seulement celles qui sont demandées.

4.2 Questions à réponses courtes (5%)

Supposons qu'il y a N définitions dans le `Dictionnaire`.

1. Combien d'opérations seront nécessaires en pire cas pour exécuter la méthode :
   a) `getDefinition(String mot)` ?

   b) `getDefinition(int index)` ?

2. Si on remplaçait l'`ArrayList<Paire<String, String>>` par une `LinkedList<Paire<String, String>>` (donc une liste chaînée qui contient une paire par noeud), combien d'opérations seraient nécessaires en pire cas pour exécuter la méthode :
   a) `getDefinition(String mot)` ?

   b) `getDefinition(int index)` ?

3. On pourrait améliorer la performance de notre `Dictionnaire` de plusieurs façons.
   Expliquez (très brièvement) une des façons d'accélérer la performance en pire cas de la méthode `getDefinition(String mot)`

Annexe A

```
// --- Fichier Whooperable.java ---
public interface Whooperable {
    public void whoop();
}

// --- Fichier A.java ---
public abstract class A {
    private String a;

    public void foo() {
        System.out.println("Foo A");
    }
    public String bar() {
        return "A";
    }
    public abstract void baz();
}

// --- Fichier B.java ---
public class B extends A {
    protected int b;

    @Override
    public void foo() {
        super.foo();
        System.out.println("Foo B");
    }
    @Override
    public void baz() {
        System.out.println("Baz");
    }
}

// --- Fichier C.java ---
public final class C extends B implements Whooperable {
    @Override
    public void whoop() {
        System.out.println("Gazoline");
    }
    @Override
    public void foo() {
        System.out.println(this.bar());
    }
    @Override
    public void baz() {
        super.foo();
    }
}
```

Annexe B

```
// --- Fichier Foo.java ---
import java.util.Arrays;

public class Foo {

    int[] elements;

    public Foo(int[] e) {
        elements = e;
    }

    public boolean egal(Object o) {
        return this == o;
    }

    public boolean egal(int[] tab) {
        return Arrays.equals(this.elements, tab);
    }
}
```

Annexe C

```
// -- Fichier Paire.java ---
public class Paire<A, B> {
    private A x;
    private B y;

    public Paire(A x, B y) {
        this.x = x;
        this.y = y;
    }

    public A getX() {
        return x;
    }

    public B getY() {
        return y;
    }

    public void setX(A x) {
        this.x = x;
    }

    public void setY(B y) {
        this.y = y;
    }
}
```

```
ArrayList
=================
ArrayList<E> liste = new ArrayList<>();

- liste.add(E element) : Ajoute l'élément à la fin
  de la liste

- liste.get(int index) : Retourne l'élément à l'index donné

- liste.set(int index, E element) : Remplace l'élément à
  l'index donné par l'élément element

- liste.size() : Retourne la taille de la liste
```