Examen de 1ère session - Janvier 2016 - Durée 2h

Cet énoncé comporte 6 pages.

Quelques rappels de XText et XTend sont donnés en Annexes D et E.

1. SimpleJ v0.1

On désire proposer un DSL XText/XTend, `SimpleJ`, qui simplifie l'utilisation de `Java` pour les développeurs. Dans sa première version, il va permettre de créer automatiquement les accesseurs et modificateurs (getters et setters).

```
1 class Point (var x: int, var y: int)
2
3 class Point3D (var x: int, var y: int, var z: int)
4
5 class Pixel (const point: Point, const couleur: String, hidden allume: boolean)
```

Le résultat attendu [1] en cas de sauvegarde du fichier (`ex1.simplej`) est le code Java présenté en Annexe A.
On note ici :

- le mot-clé **var**, qui introduit un attribut privé variable, donc doté d'un getter et d'un setter ;
- le mot-clé **const**, qui introduit un attribut privé constant, donc doté d'un getter uniquement ;
- le mot-clé **hidden** qui introduit un attribut privé caché, donc doté ni d'un getter ni d'un setter.

Les vérifications suivantes doivent être prises en compte :

- **warning** à la saisie des caractères : les noms des classes commencent par une majuscule ;
- **warning** à la saisie des caractères : les noms des attributs commencent par une minuscule ;
- **error** à la sauvegarde : les types d'attributs sont soit des types `Java` (boolean, double, int ou String) soit des classes définies dans le code.

Exercice 1.1. `SimpleJ` **v0.1 (10 points)**
Réalisez le DSL pour cette première version de `SimpleJ`. Déposez en ligne une 1ère archive `zip` [2] contenant :

☐ la grammaire ☐ le code générateur ☐ le code validateur ☐ vos trois sorties

2. SimpleJ v0.2

Nous allons maintenant rajouter le concept d'héritage simple et de programme principal à `SimpleJ` comme démontré dans le code suivant. Notez que le corps du programme est une `STRING` (ne pas tenter de définir sa grammaire).

```
1 class Point (var x: int, var y: int)
2
3 class Point3D (var z: int) : Point
4
5 class Pixel (const point: Point, const couleur: String, hidden allume: boolean)
6
7 program Main1
8 '
```

1. **Important :** nous vous embêtez pas avec l'indentation du code `Java` résultant tant que ce code compile.
2. Pour ne pas oublier quelque chose, cochez les cases quand c'est pris en compte.

```
Point p1 = new Point();
p1.setX(3); p1.setY(4);
Point3D p2 = new Point3D();
p2.setX(3); p2.setY(4); p2.setZ(5);
System.out.println(String.format("(%d,%d,%d)",p2.getX(),p2.getY(),p2.getZ())); // résultat : (3,4,5)
'
```

Le résultat attendu en cas de sauvegarde du fichier (`ex2.simplej`) est, pour les changements, le code `Java` présenté en Annexe B. Il est important de prendre en compte le fait qu'**on ne peut hériter que d'une classe définie dans le fichier**. Vous pourrez utiliser la notion de référence XText.

Exercice 2.1. `SimpleJv0.2` **(1 point [héritage] + 1 point [programme])**
Réalisez le DSL pour cette deuxième version de `SimpleJ`. Déposez en ligne une 2ème archive `zip` contenant :

☐ la grammaire ☐ le code générateur ☐ vos quatre sorties

3. `SimpleJ` v0.3

Nous allons maintenant rajouter les méthodes à `SimpleJ` comme dans le code suivant. Notez que les corps des méthodes sont des `STRING`s (qu'il est possible de spécifier avec des simples ou doubles quotes).

```
class Point (var x: int, var y: int)
        def atOrigin: boolean
        'return (x==0 && y==0);'

        def move(dx: int, dy: int)
        'x+=dx; y+=dy;'

        def toString: String
        'return String.format("(%d,%d)",x,y);'

class Point3D (var z: int) : Point
        def atOrigin: boolean
        'return (super.atOrigin() && z==0);'

        def move(dx: int, dy: int, dz: int)
        'super.move(dx,dy); z+=dz;'

        def toString: String
        'return String.format("(%d,%d,%d)",getX(),getY(),z);'

class Pixel (var delegate p: Point, var couleur: String, var allume: boolean)
        def toString: String
        'return String.format("%s[%s]",p,couleur);'

program Main3
'
Point p1 = new Point();
p1.setX(3); p1.setY(4);
Point3D p2 = new Point3D();
p2.setX(3); p2.setY(4); p2.setZ(5);
Pixel p3 = new Pixel();
p3.setP(p1); p3.setCouleur("#ffaa00"); p3.setAllume(false);
p3.move(3,4); // marche grace à la délégation sur p3.p
System.out.println(p3); // toString pas délégué, résultat: (6,8)[#ffaa00]
'
```

Vous noterez qu'au niveau des méthodes `SimpleJ` est plus simple que `Java`. Ainsi, il n'y a plus de type de retour `void` (*cf* `move`) et plus de parenthèses vides si une méthode n'a pas d'arguments (*cf* `atOrigin` ou `toString`).

`SimpleJ` dispose aussi du concept de délégation, avec le mot-clé **delegate** qui peut ou pas être présent entre le mot-clé d'accès (**var**, **const** ou **hidden**) et le nom de l'attribut. Lorsqu'une classe a un attribut délégué (ici le `Point p` dans `Pixel`) alors elle rend les méthodes du délégué en lui déléguant leur réalisation. Ainsi, si on appelle `atOrigin` ou `move` sur un `Pixel`, ce dernier délèguera cette opération à son point `p`.

Important : il ne faut pas déléguer une méthode que l'on (re)définit soi-même. Ainsi, `Pixel` ne délègue pas `toString` bien qu'elle soit définie dans `Point`, car elle est redéfinie dans `Pixel`. Ici vous prendrez uniquement le nom de la méthode en compte (ne vous embêtez pas avec les paramètres, on supposera même nom = même méthode).

TRES IMPORTANT : afin de ne pas augmenter la difficulté, vous ne prendrez en compte dans la délégation **QUE** les méthodes directement définies dans le délégué et donc **NI** celles héritées, **NI** celles déléguées, **NI** celles liées aux attributs (getters, setters). Ainsi, dans l'exemple, si `Point` héritait d'une classe `SuperPoint` qui définisse une méthode `m1`, alors `m1` ne serait pas disponible par délégation dans `Pixel`. De même, si `Point` avait un attribut délégué de classe `Delegué` qui définisse une méthode `m2`, alors `m2` ne serait pas non plus disponible dans `Pixel`. Vous ne vous occuperez pas non plus du cas où deux attributs délégués auraient les mêmes méthodes (vous les générerez toutes même si c'est incohérent – il faudrait en théorie faire un validateur pour cela).

Le résultat attendu en cas de sauvegarde du fichier (`ex3.simplej`) est le code `Java` présenté en Annexe C.

Exercice 3.1. `SimpleJv0.3` **(2 points [méthodes] + 6 points [délégation])**
Réalisez le DSL pour cette troisième version de `SimpleJ`. Déposez en ligne une 3ème archive `zip` contenant :

☐ la grammaire ☐ le code générateur ☐ vos quatre sorties

A. Résultat section 1

```java
// generated by CMO examen 2016
public class Point {

        private int x;
        private int y;

        public int getX() { return x; }

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

        public int getY() { return y; }

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

}
```

```java
// generated by CMO examen 2016
public class Point3D {

        private int x;
        private int y;
        private int z;

        public int getX() { return x; }

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

        public int getY() { return y; }

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

        public int getZ() { return z; }

        public void setZ(int z) { this.z = z; }

}
```

```java
// generated by CMO examen 2016
public class Pixel {

        private Point point;
        private String couleur;
        private boolean allume;

        public Point getPoint() { return point; }

        public String getCouleur() { return couleur; }

}
```

B. Résultat (partiel) section 2

```java
// generated by CMO examen 2016
public class Point3D extends Point {

        private int z;

        public int getZ() { return z; }

        public void setZ(int z) { this.z = z; }

}
```

```java
// generated by CMO examen 2016
public class Main1 {
        public static void main(String args[]) {

                Point p1 = new Point();
                p1.setX(3); p1.setY(4);
                Point3D p2 = new Point3D();
                p2.setX(3); p2.setY(4); p2.setZ(5);
                System.out.println(String.format("(%d,%d,%d)",p2.getX(),p2.getY(),p2.getZ()));
        }
}
```

C. Résultat section 3

```java
// generated by CMO examen 2016
public class Point {

        private int x;
        private int y;

        public int getX() { return x; }

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

        public int getY() { return y; }

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

        public boolean atOrigin() { return (x==0 && y==0); }

        public void move(int dx, int dy) { x+=dx; y+=dy; }

        public String toString() { return String.format("(%d,%d)",x,y); }
}
```

```java
// generated by CMO examen 2016
public class Point3D extends Point {

        private int z;

        public int getZ() { return z; }

        public void setZ(int z) { this.z = z; }

        public boolean atOrigin() { return (super.atOrigin() && z==0); }

        public void move(int dx, int dy, int dz) { super.move(dx,dy); z+=dz; }

        public String toString() { return String.format("(%d,%d,%d)",getX(),getY(),z); }
}
```

```java
// generated by CMO examen 2016
public class Pixel {

        private Point p;
        private String couleur;
        private boolean allume;

        public Point getP() { return p; }

        public void setP(Point p) { this.p = p; }
```

```
11
12          public String getCouleur() { return couleur; }
13
14          public void setCouleur(String couleur) { this.couleur = couleur; }
15
16          public boolean getAllume() { return allume; }
17
18          public void setAllume(boolean allume) { this.allume = allume;     }
19
20          public String toString() { return String.format("%s[%s]",p,couleur); }
21
22          public boolean atOrigin() { return p.atOrigin(); }
23
24          public void move(int dx, int dy) { p.move(dx, dy); }
25  }
```

```
1   // generated by CMO examen 2016
2   public class Main3 {
3           public static void main(String args[]) {
4
5                   Point p1 = new Point();
6                   p1.setX(3); p1.setY(4);
7                   Point3D p2 = new Point3D();
8                   p2.setX(3); p2.setY(4); p2.setZ(5);
9                   Pixel p3 = new Pixel();
10                  p3.setP(p1); p3.setCouleur("#ffaa00"); p3.setAllume(false);
11                  p3.move(3,4); // marche grace à la délegation sur p3.p
12                  System.out.println(p3); // toString pas délégué, résultat: (6,8)[#ffaa00]
13          }
14  }
```

D. Énumérations et options avec XText et XTend

Un **choix énuméré entre plusieurs modes** peut s'écrire en XText comme suit.

```
1   Definition:
2           mode=Mode (estOption='option')? name=ID
3   ;
4
5   enum Mode:
6           MODE1='m1' | MODE2='m2' | MODE3='m3'
7   ;
```

L'effet est de créér, dans le code `Java` généré à partir de la grammaire, une énumération `Mode`. Il sera ensuite possible en XTend, étant donné une variable `v` de type `Definition` de vérifier par exemple si `v.mode == Mode.MODE1`.

On note aussi ici la possibilité de définir un **élément optionnel** (ex : `m1 option Foo` *vs* `m2 Foo`). Pour savoir en XTend si l'on est dans un cas ou dans l'autre, on peut tester si `v.estOption==null` ou pas.

E. Éléments d'XTend

Le **transtypage** (casting) en XTend ne se fait pas comme en `Java`. Soit la grammaire XText suivante, et en XTend une variable `m` de type `M`.

```
1   M : M1 | M2 ;
2   M1 : attr1=ID attr2=ID ;
3   M2 : attr1=ID attr3=ID ;
```

Il est possible de faire `m.attr1` car `attr1` est défini dans `M1` et `M2`, et donc "présent" dans `M`. Par contre `m.attr2` (indéfini) et `((M1)m).attr2` (erreur de syntaxe) sont incorrects. En XTend on écrira `(m as M1).attr2`.

XTend propose aussi des **constructions très riches pour les collections**. Soit promo une variable contenant une collection etudiants de Etudiant, avec name et age des attributs de Etudiant (voir ci-dessous la grammaire). Alors on a :

```
promo.etudiants                                  // étudiants
promo.etudiants.map[name]                        // noms des étudiants
promo.etudiants.map[name].join(", ")             // String des noms séparés par des virgules
promo.etudiants.filter[age>=18]                  // étudiants majeurs
promo.etudiants.filter[age>=18].size             // nombre d'étudiants majeurs
promo.etudiants.filter[it.addresse.ville="Paris"].size  // nombre d'étudiants vivant à Paris
if(promo.etudiants.exists[age<18]) { ... }       // s'il existe au moins un étudiant mineur ...
```

XTend dispose de plusieurs méthodes pour construire des collections à partir d'une itération :

```
promo.etudiants.map[name]          // noms des étudiants (itérateur)
promo.etudiants.map[name].toList   // la liste qui correspond
promo.etudiants.map[name].toSet    // l'ensemble qui correspond
promo.etudiants.toMap[name]        // le tableau associatif Map<String,Etudiant>, it.name -> it
```

Enfin, un dernier exemple. Soit la grammaire :

```
Promotion : name=ID '=' etudiants+=Etudiant (',' etudiants+=Etudiant)*;
Etudiant : name=ID ':' age:INT
```

Soit le modèle suivant :

```
MIAGE = Alice:18, Bob:16, Charles:20, Daniel:21
```

Pour générer ceci (les majeurs) :

```
MIAGE {
        - Alice a 18 ans;
        - Charles a 20 ans;
        - Daniel a 21 ans
}
```

on peut écrire (remplacer << par « et >> par ») :

```
def doGenerate(Promo p) '''
        <<p.name>> {
        <<p.etudiants.filter[age>=18].map[doGenerateEtudiant].join(";\n")>>
        }
'''
def doGenerateEtudiant(Etudiant e) '''
        - <<e.name>> a <<e.age>> ans
'''
```

Bien sur, on peut aussi le faire (modulo le ; sur la dernière personne) avec les instructions de contrôle de XTend (IF ... ENDIF et FOR ... ENDFOR) et la notion de receveur implicite (a.doGenerateA == doGenerateA(a)) :

```
def doGenerate(Promo p) '''
        <<p.name>> {
        <<FOR etudiant : p.etudiants>>
        <<IF etudiant.age>=18>>
                - <<etudiant.doGenerateEtudiant>> ;
        <<ENDIF>>
        <<ENDFOR>>
        }
'''
```

ou bien encore d'autres façons.