

# INTRODUCTION A LA PROGRAMMATION

## Test Semestre I

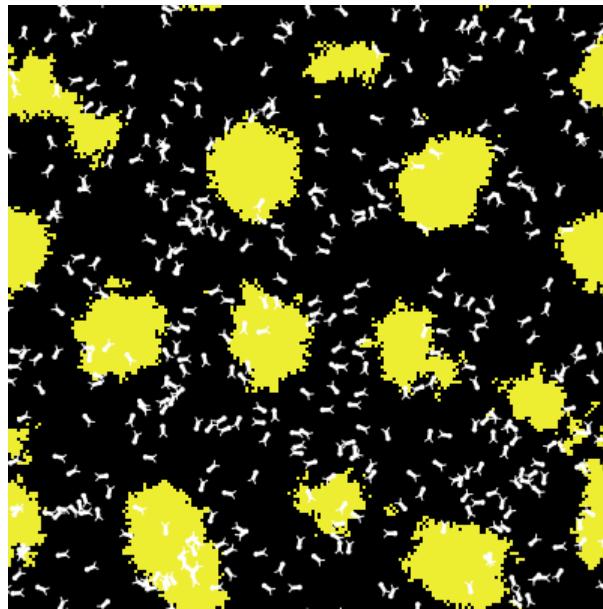
### Instructions :

- Vous disposez de une heure quarante cinq minutes pour faire cet examen (13h15 - 15h) ;
- Nombre maximum de 115 points (dont environ 25 facultatifs) ;
- Toute documentation sur papier est autorisée ;
- Veillez à **ne traiter qu'un exercice par feuille**, et à **indiquer votre numéro sciper** sur *chacune* des feuilles. Une feuille sans identification ne sera pas corrigée ;
- L'examen compte 3 exercices. Ces exercices sont indépendants.
- Les exercices ne sont pas tous de même difficulté. Commencez par ceux dont vous maîtrisez le mieux les concepts impliqués.

**Suite au verso ↗**

## Exercice 1 : Conception OO [ 55 points]

On s'intéresse ici à simuler graphiquement la propagation de termitières dans un environnement jonché de brindilles de bois.



Une partie du programme est fournie en annexe ; à savoir :

- une modélisation minimale de l'environnement à simuler `Environment` ;
- une classe `Positionable` ;
- une interface `Updatable` ;
- un type énuméré `MovingMode`.

**Prenez connaissance de ce matériel avant de commencer l'exercice et ne négligez pas les commentaires qui y figurent.** Notez que nous ne nous préoccupons pas dans cet exercice de la représentation graphique de la simulation.

Les *termitières* sont des amas de *brindilles* de bois peuplées de *termites*.

Pour simplifier on considérera qu'une termitière a une position unique qui la caractérise : pour savoir à quelle termitière unique appartient une terme, il suffit donc d'en connaître la position.

Une brindille de bois a une position dans l'environnement et est caractérisée par sa *longueur*. La position et la longueur sont initialisées à la construction.

Une terme est également positionnée dans l'environnement. Elle est caractérisée par un *niveau d'énergie* et doit être considérée comme un `Updatable`.

La construction d'une terme prendra en paramètre la position de la termitière à laquelle elle appartient (qui coïncidera alors avec sa propre position). Elle naîtra avec un niveau d'énergie donné.

Elle peut se déplacer au hasard ou en ciblant un point précis (un `Vector` tel qu'utilisé dans votre projet). Les modalités précises de mises en oeuvre de ces modes de déplacement ne nous intéressent pas ici. Le seul élément à prendre en compte est que le déplacement ciblé a besoin de connaître la position ciblée.

Le comportement général d'une terme est le suivant : elle *se déplace au hasard* jusqu'à rencontrer une brindille de bois. Lorsqu'elle en rencontre une, elle la prend et *se déplace alors en mode ciblé* avec pour objectif la position de sa termitière. Une fois arrivée à sa termitière elle dépose la brindille et se déplace à nouveau au hasard. A chaque déplacement, la terme perd une quantité donnée d'énergie. A chaque fois qu'elle rejoint sa termitière, elle se restaure et gagne de l'énergie. Les quantités d'énergie perdue et

gagnée sont constantes et identiques pour toutes les termites (par exemple 0.5 pour celle perdue et 1 pour celle gagnée). Si au cours d'un déplacement aléatoire la termite voit son énergie baisser en dessous d'un seuil donné (un nombre identique pour toutes les termites, par exemple valant 20.0), elle se fixera pour objectif de retourner à sa termitière et adoptera alors à nouveau un mode de déplacement ciblé vers cet objectif.

**Termites sexuées** Certaines termites peuvent être sexuées. Elles ont dans ce cas un genre (mâle ou femelle). Les termites sexuées peuvent migrer et changer de termitière. Elles peuvent, comme les termites asexuées, se déplacer au hasard ou en ciblant un point précis. Pour simplifier, on supposera que les modalités sont identiques (dans le cas sexué ou asexué) pour chaque type de déplacement. Le comportement des termites sexuées est le suivant : si elles ne sont pas en train de migrer, elles se comportent comme les termites asexuées. Sinon, elles ciblent le point destination de leur migration.

**Reproduction, migration et fin de vie** A chaque appel de la boucle de simulation, une termitière peut faire naître au hasard des termites sexuées et des termites asexuées. Si le peuplement de la termitière atteint un certain seuil (donné par une constante commune à toute les termitières), toutes les termites sexuées quittent la termitière (migration). Elles se fixent alors comme cible un point au hasard ; le même point pour toutes. Lorsqu'elles arrivent à ce point elles se posent. Les termites qui ont migré font alors partie de la termitière située à cet emplacement ; qui doit être créée s'il elle n'existe pas. Vous considérerez qu'une termite ne change son appartenance à une termitière qu'une fois atteint la position cible de sa migration. Une termite, quel que soit son type, meurt si son niveau d'énergie devient nul. Elle disparaît alors de la simulation.

Le but est donc de pouvoir simuler l'évolution des termitières.

### Question 1 : Conception [45 points]

On vous demande d'écrire la/les classe(s) permettant de compléter les éléments manquants et permettant de mettre en oeuvre les fonctionnalités souhaitées ; seules les entêtes des méthodes sont demandés. **On ne vous demande pas d'écrire de programme complet, ni le corps des méthodes, uniquement les attributs et les entêtes de méthodes.** Vous pouvez décrire les classes et leur contenu au moyen de diagrammes ou en pseudo-code Java. Les contraintes suivantes seront respectées :

1. vous supposerez notamment qu'un programme principal existe, qu'il crée un objet `Environment`, et appelle en boucle sa méthode `simulate`. Cette méthode permet de faire évoluer les entités impliquées au cours du temps (au moyen de méthodes `update`, telle que dictées par l'interface `Updatable`) ;
2. les classes devront contenir les membres nécessaires pour mettre en oeuvre toutes les fonctionnalités souhaitées, **qu'elles soient explicitement demandées ou suggérées par les spécifications de l'énoncé** (en particulier par la mise en oeuvre des méthodes `update`) ;
3. Pour les méthodes pour lesquelles ce n'est pas clair **vous noterez au moyen d'un bref commentaire la signification du type de retour et ce qu'elles font dans les grandes lignes**.
4. **Ne négligez ni les constructeurs, ni les droits d'accès et les modificateurs.**
5. le droit `protected` **ne doit pas être utilisé** pour les attributs ;
6. votre conception ne doit pas nécessiter de dupliquer du code ;
7. vous ne proposerez de getter/setter que lorsqu'ils sont nécessaires à la mise en oeuvre de la simulation décrite ;
8. il n'est pas nécessaire d'écrire des directives d'importation.

**Suite au verso ↗**

---

**Question 2 : Programmation [10 points]**

Au vu de votre conception, donnez le corps de la méthodes `update` des termites sexuées. Il n'est pas nécessaire de donner le corps des méthodes utilisées provenant des autres classes.

## Exercice 2 : Concepts [37 points]

Répondez clairement et succinctement aux questions suivantes :

1. [5 points] Soit le code suivant :

```

0. class X {
1.     protected int x;
2.
3.     public X() {
4.         this(100);
5.     }
6.     public X(int x) {
7.         this.x = x;
8.     }
9. }
10.
11. class Y extends X {
12.     protected int y;
13.
14.     public Y(int x, int y) {
15.         this.x = x;
16.         this.y = y;
17.     }
18. }
```

- (a) Quels constructeurs sont invoqués lors de la création d'une instance de Y ?  
 (b) Quelle(s) critique(s) pouvez vous formuler sur ce code ? comment y remédier ?
2. [ 4 points] Qu'affiche le code suivant :

```

0. class Question {
1.     public static void main(String[] args) {
2.         String s1 = new String("Ragging");
3.         String s2 = new String("Bull");
4.
5.         p(s1,s2);
6.
7.         System.out.println(s1);
8.         System.out.println(s2);
9.
10.    }
11.
12.
13.    public static void p(String s1, String s2) {
14.        String tmp;
15.        tmp = s1;
16.        s1 = s2;
17.        s2 = tmp;
18.    }
19. }
```

Justifiez brièvement votre réponse.

**Suite au verso ↗**

3. [7 points] Soit le code suivant :

```

0. interface I {}
1.
2. interface J extends I {}
3.
4. abstract class A {}
5.
6. class B extends A implements J {}
7.
8. class Question {
9.     public static void main(String[] args) {
10.    }
11. }
```

Pour chacune des instructions suivantes, mises séparément dans `main`, indiquez si elles compilent ou non :

- (a) `A o = new A()`
- (b) `A o = new B();`
- (c) `B o = (I) new B();`
- (d) `A o = new I();`
- (e) `I o = new B();`
- (f) `J o = new B();`
- (g) `J o = (B) new J();`

Justifiez brièvement.

4. [4 points] La compilation du code suivant :

```

0. class Question {
1.     public static void main(String[] args) {
2.         approx(10.);
3.         approx(0.);
4.     }
5.
6.     public static double approx(double x) {
7.         if (x<= 0) throw new Exception("NaN");
8.         return Math.log(x);
9.     }
10. }
```

cause ce message d'erreur :

`unreported exception Exception; must be caught or declared to be thrown`

- (a) expliquez à quoi cela est dû ?
- (b) comment corrigeriez-vous ce problème ? (indiquez ce que vous ajouteriez au code et à quel(s) endroit(s))

5. [ 3 points] Une méthode abstraite peut-elle être finale ? Justifiez brièvement.

6. [6 points] Soit le code suivant :

```

0. interface Drawable {
1.     default void draw() {}
2.     void draw(String color);
3. }
4.
5. class Ball implements Drawable {
6.     protected void draw(String color) {}
7. }
8.
9. class Question {
10.    public static void main(String[] args) {
11.        Ball ball = new Ball();
12.        Ball.draw();
13.    }
14. }
```

- (a) Pourquoi sa compilation cause t-elle le message d'erreur suivant :  
`attempting to assign weaker access privileges; was public`
- (b) Pourquoi sa compilation cause t-elle le message d'erreur suivant :  
`non-static method draw() cannot be referenced from a static context`
- (c) Y a-t-il de nouvelles fautes de compilation si :
  - i. on supprime la déclaration de la méthode `draw` dans `Ball` (ligne 6)
  - ii. si on remplace la ligne 11 par `Drawable ball = new Ball();`
 Justifiez brièvement.

7. [ 4 points] Pourquoi la tournure :

`List<Double> myList = new ArrayList<>();`

est-elle généralement jugée préférable à celle-ci :

`ArrayList<Double> myList = new ArrayList<>();`

**Suite au verso ↗**

8. [4 points] Soit le programme suivant :

```
0. class Account {
1.     private double amount;
2.     public double setAmount(double m) {
3.         amount = m;
4.         return m;
5.     }
6. }
7.
8. class Bank {
9.     List <Account> accounts = new ArrayList<>();
10.
11.    public Bank(List<Account> accounts) {
12.        this.accounts = accounts;
13.    }
14. }
```

- (a) Si l'on part de l'hypothèse qu'un compte (**Account**) peut exister en dehors d'une banque, mais qu'une fois assigné à cette dernière il doit en devenir la propriété, quelle faille d'encapsulation contient ce code et comment la réparer ? (donnez simplement une explication en français)
- (b) quelle changement proposeriez-vous pour que le concept de compte n'existe pas sans celui de banque ? (donnez une brève explication en français).

## Exercice 3 : Déroulement de programme [23 points]

Le programme suivant compile et s'exécute sans erreurs.

```

0. interface I {
1.     default int m1() { return 10; }
2.     default int m2() { return 5; }
3. }
4. abstract class A {
5.     private int a;
6.     private int b;
7.
8.     public A(int x, int y) { a=x; b=y; }
9.
10.    public A(int x) { this(x,3); }
11.    public A() { this(2); }
12.
13.    public int m1() { return 2*a; }
14.
15.    public abstract int m2();
16.
17.    public int m3() {
18.        return getB() + m1() + m2();
19.    }
20.    public int getA() { return a; }
21.    public int getB() { return b; }
22. }
23. class B extends A {
24.     private int b;
25.
26.     public B(int x, int y) {
27.         super(x,y);
28.         b = 3;
29.     }
30.     public int m1() {
31.         return getB()*getA();
32.     }
33.     public int m2() {
34.         return 5*super.getB();
35.     }
36.     public int getB() { return b; }
37. }
38.
39. class C extends B {
40.     private int c;
41.     public C(int x, int y, int z) {
42.         super(x,y);
43.         c=z;
44.     }
45.     public int m1() {
46.         return 10*getA();
47.     }
48.     public int m2() {
49.         return 3*getB()+c;
50.     }
51. } // fin de la classe C
52. class D extends B implements I {
53.     public D() {
54.         super(3,4);
55.     }
56.     public int m1() {
57.         return super.m1() + I.super.m1();
58.     }
59.     public int m2(A a, B b) {
60.         return a.m1() + b.m2();
61.     }
62. }
63. class Deroulement {
64.     public static void main(String[] args) {
65.         int k=0;
66.         B b = new B(3,5);
67.         A a = b;
68.         System.out.println( ++k + " " + b.m1());
69.         System.out.println( ++k + " " + b.m2());
70.         System.out.println( ++k + " " + a.m1());
71.         System.out.println( ++k + " " + a.m2());
72.         System.out.println( ++k + " " + a.m3());
73.         System.out.println( "||||||" );
74.         C c = new C(3,4,5);
75.         a = c;
76.         b = c;
77.         System.out.println( ++k + " " + c.m1());
78.         System.out.println( ++k + " " + c.m2());
79.         System.out.println( ++k + " " + a.m1());
80.         System.out.println( ++k + " " + a.m2());
81.         System.out.println( ++k + " " + a.m3());
82.         System.out.println( ++k + " " + b.m1());
83.         System.out.println( ++k + " " + b.m2());
84.         System.out.println( ++k + " " + b.m3());
85.         System.out.println( "||||||" );
86.         D d = new D();
87.         a = d;
88.         b = d;
89.         System.out.println( ++k + " " + d.m1());
90.         System.out.println( ++k + " " + d.m2());
91.         System.out.println( ++k + " " + d.m2(a,b));
92.         System.out.println( ++k + " " + d.m2(a,c));
93.         System.out.println( ++k + " " + d.m2(c,c));
94.         System.out.println( ++k + " " + a.m3());
95.     }
96. }

```

Qu'affiche t-il ? Expliquez succinctement son déroulement. Il ne s'agit pas ici de paraphraser le code, mais bien d'*expliquer* les étapes et le déroulement du programme.