# 2i002 – Examen final noté sur 60 pts

Durée : 2 heures – Aucun document, pas de calculatrice, barème donné à titre indicatif

---

**Exercice 1 – Encore des problèmes d'égalité... Entre autres (14pts)**

```
1 public class Pion{
2   protected int id;
3   public Pion(int id){
4     this.id = id;
5   }
6   public boolean equals(Pion p){
7     return id == p.id;
8   }
9 }
10
11 public class PionNomme extends Pion{
12   private String n;
13   public PionNomme(String n, int id){
14     super(id);
15     this.n = n;
16   }
17 }
```

```
1 public class TestEqualsPion{
2   public static void main(String[] args){
3     Pion p1 = new Pion(3); Pion p2 = new Pion(3);
4     Object o1 = p2; Object o2 = 3;
5     PionNomme pn1 = new PionNomme("toto", 3);
6     Pion pn2 = new PionNomme("titi", 3);
7
8     System.out.println(p1.equals(p2));
9     System.out.println(p1.equals(o1) + "␣" + o1.equals(p1));
10    System.out.println(p2.equals(o1) + "␣" + o1.equals(p2));
11    System.out.println(p2.equals(o2) + "␣" + o2.equals(p2));
12    System.out.println(pn1.equals(pn2) + "␣" + pn2.equals(pn1));
13    System.out.println(p1.equals(pn1) + "␣" + pn1.equals(p1));
14  }
15 }
```

```
16 public class TestCastPion{
17   public static void main(String[] args){
18     Object o1 = 2;
19     o1 = new Pion(2);
20     PionNomme pn1 = new Pion(3);
21     PionNomme pn2 = (PionNomme) new Pion(4);
22     Pion p1 = new PionNomme("titi", 5);
23     Pion p2 = (Pion) new PionNomme("titi", 6);
24     Object o2 = new PionNomme("titi", 7);
25     Pion p3 = o2;
26     Pion p4 = (Pion) o2;
27     Pion p5 = (PionNomme) o2;
28   }
29 }
```

```
30 public class TestComptePion{
31   public static void main(String[] args){
32     Pion[] tab = new Pion[10];
33     Poin p = new Pion(12);
34     for(int i = 0; i<tab.length; i++){
35       if(i % 4 == 0) p = new PionNomme("toto", i);
36       else if(i % 2 == 0) p = new Pion(i);
37       else p = tab[i-1];
38       tab[i] = p;
39     }
40   }
41 }
```

**Aide** : $i\%4 == 0$ pour i = 0, 4, 8, ... ; $i\%2 == 0$ pour i = 0, 2, 4, 6, 8, ...

**Q 1.1 (2.5pts)** Dans `TestComptePion`. Combien d'instances de `Pion` et de `PionNomme` ont-elles été créées ? Combien en reste-t-il à l'issue de l'exécution de la boucle `for` (arrivé entre les lignes 39 et 40) ?

1.5pt pour 3+3 instances créées
1pt pour les 3+2 instances restantes à la fin
case 0 : PionNomme
case 1 : même instance que case 0
case 2 : Pion
case 3 : même instance que case 2
...
3 instances de PionNomme
3 instance de Pion (avec la première)
5 simples références
A la fin de l'exécution, il reste 5 instances (3 PionNomme) et 2 Pion

**Q 1.2 (1.5pt)** Nous déplaçons maintenant la ligne 32 (instanciation du tableau) entre les 34 et 35 (première ligne de code dans la boucle `for`)... Le même nombre d'instances est créé... Mais combien en reste-t-il arrivé entre les lignes 39 et 40 ? Justifier rapidement.

Si les étudiants répondent 1 (=Pion(12) ) ⇒ 0
Si les étudiants répondent 1 (= PionNomme(8)) ⇒ 1 (même si c'est faux, c'est quand même pas bête)

1.5 s'ils répondent 0 avec un semblant de justification.
C'est une question piège : une fois sur deux p pointe vers null... La solution n'est donc pas forcément 1...
la dernière itération correspond à $i = 9$ donc :

```
p = tab[i-1] // =null
```

⇒ il reste 0 instance à la fin

**Q 1.3 (2pts)** Dans `TestCastPion` nous avons testé différents cas de figure de conversion de type de variables (cast) ; indiquer les lignes qui posent problème (KO1 pour un problème de compilation ou KO2 pour un problème à l'exécution)

-0.5 par faute
-0.25 par faute de type de KO

```
Object o1 = 2;
o1 = new Pion(2); // OK
PionNomme pn1 = new Pion(3); // KO1
PionNomme pn2 = (PionNomme) new Pion(4); // KO2
Pion p1 = new PionNomme("titi", 5); // OK
Pion p2 = (Pion) new PionNomme("titi", 6); // OK
Object o2 = new PionNomme("titi", 7); // OK
Pion p3 = o2; // KO1
Pion p4 = (Pion) o2; //OK
Pion p5 = (PionNomme) o2; //OK
```

**Q 1.4 (1pt)** La classe `Pion` est fonctionnelle. Le code `TestEqualsPion` ci-dessus est-il fonctionnel ? Rappel : un code est fonctionnel quand il compile et s'exécute sans erreur.

Oui, aucun soucis sur les equals de o2 qui vont chercher le code de equals dans Object

**Q 1.5 (3pts)** Donner les affichages issus du `main` de `TestEqualsPion` et expliquer succinctement le mécanisme de sélection de méthode dans chaque cas de figure. Si certaines lignes posent problème : indiquer la nature du problème et poursuivre l'exécution virtuelle du code comme si la ligne avait été commentée.

0.5pt pour p1/p2
1pt pour o1/p1
1pt pour p2/o1
0.5 pour p2/o2
0.5pt pour pn1/pn2
0.5 pour p1/pn1
⇒ 0 si pas de détail sur la présélection de méthode / identification de la présence d'equals dans `Object`. On n'est pas trop vache sur les justifications : dès lors que la bonne méthode est indiquée, c'est bon.

```
// ① equals de Object
// ② equals de Pion
    Pion p1 = new Pion(3); Pion p2 = new Pion(3);
    Object o1 = p2; Object o2 = 3;
    PionNomme pn1 = new PionNomme("toto", 3);
    Pion pn2 = new PionNomme("titi", 3);

    System.out.println(p1.equals(p2) + "␣" + p2.equals(p1)); // true ②
    System.out.println(p1.equals(o1) + "␣" + o1.equals(p1)); // false, false ① (argument non
        dynamique), ① (truc dur de la présélection du compilateur)
    System.out.println(p2.equals(o1) + "␣" + o1.equals(p2)); // true, true, ①, ① même instance
    System.out.println(p2.equals(o2) + "␣" + o2.equals(p2)); // / false, false ①, ① ⇒ pas la même
        instance
    System.out.println(pn1.equals(pn2) + "␣" + pn2.equals(pn1)); // true ②, true ② un PionNomme
        est un Pion
    System.out.println(p1.equals(pn1) + "␣" + pn1.equals(p1)); // true ②, true ② un PionNomme est
        un Pion
```

**Q 1.6 (2pts)** Cette version de l'égalité n'étant pas satisfaisante, donner une version correcte de la fonction `equals` dans `Pion`.

⇒ Il faut être sévère, j'ai répété plusieurs fois que ça tombait à l'exam
2 si tout est juste (0 si n'importe quelle faute –cast, oubli du test null, instaceof au lieu de getClass....–). Evidemment, on peut nuancer sur les petites fautes de syntaxe.

```
1 boolean equals(Object o){
2   if(o == null) return false;
3   if(o == this) return true; // OPT
4   if(o.getClass() != getClass()) return false;
5   Pion p = (Pion) o;
6   return p.id == id;
7 }
```

**Q 1.7 (2pts)** Est-il nécessaire de développer une version spécifique de `equals` pour `PionNomme` ? Dans l'affirmative donner le code nécessaire (uniquement les lignes qui seraient différentes de la réponse précédente).

1 : OUI c'est nécessaire + code juste
1 pour equals entre String

```
1 PionNomme p = (PionNomme) o;
2 return p.id == id && p.n.equals(n); // id protected, pas de soucis
```

## Exercice 2 – Chasse au trésor (10 pts)

```
1 public class MainTresor {
2       ①
3     public static void main(String[] args) {
4         int dim = Integer.valueOf(args[0]);
5         carte =  ②;
6         for(int i=0; i<dim; i++)
7             for(int j= 0; j<dim; j++)
8                 carte[i][j] = dim;
9         // position du tresor
10        int x = (int) (Math.random()*dim);
11        int y = (int) (Math.random()*dim);
12        // marquage de la position
13        carte[x][y] = 0;
14              ③
15            while(true){
16                x = (int) (Math.random()*dim);
17                y = (int) (Math.random()*dim);
18                testCase(x, y);
19            }
20              ④
21    }
22        ⑤
23 }
```

**Q 2.1 (1.5pt)** Déclarer la variable `carte` au niveau de ① et instancier la au niveau de ② en utilisant la dimension `dim` récupérée à la ligne 4.

```
1 ① private static int[][] carte;
2 ② carte = new int[dim][dim];
```

1pt pour la déclaration (-0.5 si pas de private, 0 si pas de static et/ou des choses exotiques dans les crochets)
0.5pt pour l'instanciation

**Q 2.2 (2.5pts)** Nous avons décidé de gérer la découverte du trésor avec des exceptions... L'idée est la suivante : nous allons tester des coordonnées aléatoires dans une boucle (lignes 15-19) et la méthode `testCase` lèvera une exception lorsque le trésor est découvert, ce qui permettra de sortir de la boucle.
Donner le code de la classe `ExceptionMessageCoord` qui doit permettre de stocker/transporter les informations relatives à la position du trésor (`x, y`) ainsi qu'un message indiquant la découverte (`"Trésor TROUVE !!!"`). Ce message sera accessible via la méthode `getMessage` qui existe dans `Exception` et qui renvoie la chaine passée en argument du constructeur d'`Exception`.

0.5 pour la signature de classe
1 pt pour les attributs (0.5 si la déclaration est autre que public)
1pt pour le constructeur (0 si oubli des arguments ou erreur dans le super)

```
1 public class ExceptionMessageCoord extends Exception {
2     public int x,y;
3
4     public ExceptionMessageCoord(int x, int y, String message) {
5         super(message);
6         this.x = x;
7         this.y = y;
8     }
9 }
```

**Q 2.3 (2pts)** Donner le code de la fonction `testCase` qui lève une exception lorsque le trésor est découvert (c'est à dire lorsque les coordonnées `x,y` correspondent à un 0 dans la `carte`). Cette fonction est implémentée à la position ⑤. Vous attacherez une attention particulière à la signature de la méthode pour plusieurs raisons.

1pt pour la signature (0.5 static, 0.5 throws)
1pt pour le corps de la méthode (0.5 throw, 0.5 construction d'une exception cohérente)

```
1     public static void testCase(int x, int y) throws ExceptionMessageCoord{
2         if(carte[x][y] == 0)
3             throw new ExceptionMessageCoord(x, y, "Trésor␣trouvé␣!");
4     }
```

**Q 2.4 (2pts)** Donner le code à insérer aux positions ③ et ④ pour gérer l'exception levée par `testCase` et afficher les coordonnées du trésor lorsqu'il est découvert.

```
1 try{ // ③
2   while(true){
3                 x = (int) (Math.random()*dim);
4                 y = (int) (Math.random()*dim);
5                 testCase(x, y);
6             }
7 }catch(ExceptionMessageCoord e){ // ④
8             System.out.println(e.getMessage() + e.x +"␣"+e.y)
9 }
```

**Q 2.5 (2pts)** Donner les lignes pour compiler les classes et exécuter ce `main` sur une carte de 10 cases de coté. Quel est le comportement général du programme si la dimension est invalide, quelle ligne pose problème ? Donner un exemple d'appel provoquant le problème.

0.5 compilation
0.5 appel avec argument
0.5 le problème est la levée d'une exception à la ligne 4 (rupture de calcul)
0.5 pour l'exemple

```
1 javac *. java OU javac ExceptionMessageCoord.java MainTresor.java
2
3 java MainTresor 10
4
5 java MainTresor dlkfgj OU java MainTresor
```

## Exercice 3 – Plus blanc que blanc (10pts)

Dans l'optique d'une simulation de lavage du linge, nous allons travailler sur les éléments suivants : `MachineALaver`, `Linge`, `Pantalon`, `Jean`, `TShirt`, `Pull`, `SousVetement`. Tout linge peut être propre ou sale. Concernant le repassage, nous considérons que les sous-vêtements sont toujours repassés alors que les autres éléments peuvent être froissés ou repassés. Tout linge possèdera donc une méthode `boolean estPropre()`, une méthode `boolean estRepasse()` (qui renvoie toujours `true` pour les `SousVetement`), des méthodes `void laver()` et `void salir()`. Les méthodes `void froisser()` et `void repasser()` ne seront présentes que lorsque c'est nécessaire. Pour faire fonctionner tout cela facilement, nous introduirons deux attributs booléen `propre` et `repasse`.
**Note :** vous pouvez répondre à toutes les questions suivantes d'un seul coup si vous le souhaitez.

**Q 3.1 (1pt)** Donner une hiérarchie simple de ces classes en indiquant si elles sont abstraites ou pas.

```
1 Linge <<ABS>>
2         Pantalon <<ABS>> ou pas, comme on veut
3                 Jean
4         TShirt
5         Pull
6         SousVetement
7
8 La MachineALaver doit être à part. Elle contient un tableau de Linge
```

⇒Les étudiants peuvent directement donner la hiérarchie complétée s'ils le souhaite

**Q 3.2 (1pt)** Il intéressant d'ajouter une classe intermédiaire dans la hiérarchie pour mieux gérer les fonctions dédiées au repassage. Proposer un nom de classe et une intégration dans la hiérarchie précédente.

Il est intéressant d'ajouter une classe Vetement (du dessus) :

```
1 Linge <<ABS>>
2         Vetement <<ABS>>
3                 Pantalon <<ABS>> ou pas, comme on veut
```

```
4                           Jean
5                   TShirt
6                   Pull
7           SousVetement
```

**Q 3.3 (3pts)** Indiquer dans quelles classes doivent être stockés le ou les attributs et la ou les méthodes relatives aux lavage et repassage. Vous indiquerez clairement si les méthodes sont concrètes ou abstraites. Afin de rendre l'architecture plus intéressante, nous imposons que les deux attributs `propre` et `repasse` soient stockés dans deux classes différentes.

0.5pt pour les attributs/méthodes de Linge
+ 1pt pour estRepasse() << *ABS* >>
1pt pour la gestion du repassage dans Vetement
0.5 pour penser à donner une implémentation de estRepasser dans SousVetement

```
Linge <<ABS>>
 - boolean propre
 + estLave()
 + estRepasse() <<ABS>>
 + laver()
        Vetement <<ABS>>
          - boolean repasse
          + repasser()
          + froisser()
          + estRepasse()

                Pantalon <<ABS>> ou pas, comme on veut
                        Jean
                TShirt
                Pull

        SousVetement
          + estRepasse()
```

Note, il est impossible de gérer le repassage avec un attribut dans Linge et sans méthode abstraite avec la contrainte supplémentaire...

**Q 3.4 (2pts)** Nous avons choisi de construire la classe `MachineALaver` comme une classe fille de `ArrayList<Linge>`. Donner le code **minimum** de la classe `MachineALaver` permettant de construire l'objet, d'ajouter du linge et de compter les objets présents dans la machine. **ATTENTION**, je veux le code **minimum** pour réaliser ces opérations, exploitant toutes les méthodes héritées qu'il n'est pas nécessaire de redéfinir. La doc `ArrayList` est disponible à la fin du sujet.

Il suffit de donner :

```
public class MachineALaver extends ArrayList<Linge>{
}
```

On ne sanctionne pas s'il ajoute un import (évidemment), ni s'ils ajoutent un constructeur (c'est toujours mieux de le voir)... Par contre, on met -1 dès qu'ils redéfinissent des add, get ou size. On met 0 s'il y a un attribut ArrayList.

**Q 3.5 (3pts)** Dans la classe `MachineALaver`, donner le code de la méthode `void lavage()` qui lave tout le linge et froisse le linge qui est froissable.

1pt pour le for dans this
1pt pour le instanceof
1pt pour le cast avant froisser

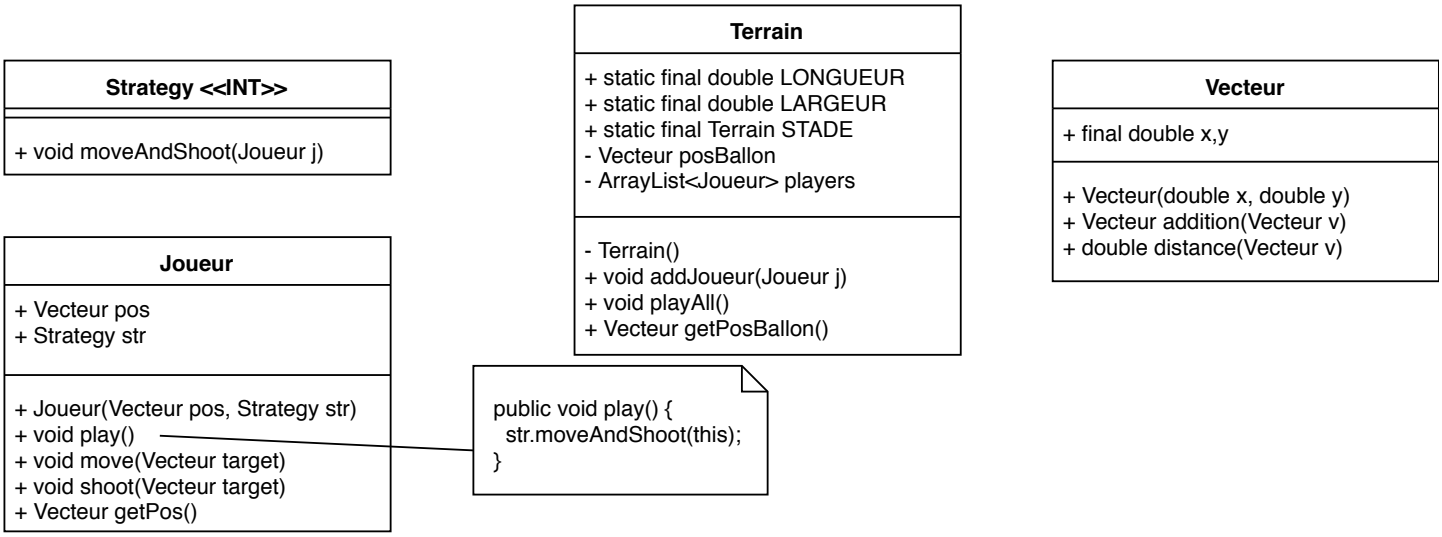
```
// Dans MachineALaver

public void lavage(){
  for (Linge li : this){
    li.laver();
```

```
6     if(li instanceof Vetement)
7       ((Vetement) li).froisser();
8   }
9 }
```

## Exercice 4 – Football & stratégies (9 pts)

Nous disposons des classes suivantes (rappel + : public, − : private, << $INT$ >> désigne une interface, les autres classes sont conrètes) :




**Q 4.1 (1.5pt)** La classe `Terrain` a été conçue en Singleton. Indiquer les commandes pour créer ou récupérer une instance de `Terrain` depuis un `main`. Indiquer brièvement s'il est possible de créer plusieurs instances de `Terrain` et, dans l'affirmative, comment ?

0.5pt ligne d'appel
1pt repérage du constructeur privé

```
1 Terrain t = Terrain.STADE;
```

Il n'est pas possible de créer d'autres instances (le constructeur est privé)

**Q 4.2 (1pt)** Donner le code de la méthode `playAll()` de la classe `Terrain` qui fait *jouer* tous les joueurs présents sur le terrain.

```
1 public void playAll(){
2   for(Joueur j:players)
3     j.play();
4   }
```

**Q 4.3 (3pts)** Donner le code d'une classe `StrategyBallon` consistant à foncer vers le ballon et à shooter dans une direction aléatoire [1] si le ballon est à moins d'un mètre du joueur.

0.5pt pour la signature classe
1 pt pour j.move
1.5 pt pour le reste

1. Vous construirez une cible aléatoire dans les coordonnées du terrain.

```java
public class StrategyBallon implements Strategy{
  // Pas besoin de constructeur... Mais ce n'est pas faux d'en mettre un
  public void moveAndShoot(Joueur j){
    j.move(Terrain.STADE.getPosBallon());
    if(j.getPos().distance(Terrain.STADE.getPosBallon())<1)
      j.shoot(new Vecteur(Math.random() * Terrain.LARGEUR, Math.random() * Terrain.LONGUEUR));
  }
}
```

**Q 4.4 (3.5pts)** Nous imaginons maintenant que nous disposons de deux `Strategy` : `StrategyAttaque` et `StrategyDefense`. Nous voulons construire un joueur qui joue en *défense* lorsque le ballon est en-deça de la moitié de terrain (coordonnée `y` du ballon entre `0` et `LONGUEUR/2`) et en *attaque* dans le cas contraire. Quelle(s) classe(s) faut-il ajouter ? Donner le code de cette ou ces classes et indiquer les lignes permettant de créer un tel joueur dans le `main` (sans vous occuper de sa position initiale).
Note : on rappelle qu'il est interdit de modifier une classe fonctionnelle.

Il faut faire une nouvelle stratégie mais ne surtout pas toucher au joueur...
2pts pour l'archi d'une stratégy en contenant 2
1.5pt pour la réalisation

```java
public class StrategyLibero implements Strategy{
  private Strategy def, att;

  public StrategyLibero(){
    def = new StrategyDefense();
    att = new StrategyAttaque();
  }

  public void moveAndShoot(Joueur j){
    if(Terrain.STADE.getPosBallon().y < Terrain.LONGUEUR/2)
        def.moveAndShoot(j);
    else
        att.moveAndShoot(j);
  }
}
```

**Rappel de documentation :** `ArrayList<Object>`

Il faut ajouter la commande `import java.util.ArrayList;` en début de fichier pour utiliser les `ArrayList`.

- Instanciation : `ArrayList<Object> a = new ArrayList<Object>();`
- `void add(Object o)` : ajouter un élément à la fin
- `Object get(int i)` : accesseur à l'item `i`
- `Object remove(int i)` : retirer l'élément et renvoyer l'élément à la position `i`
- `int size()` : retourner la taille de la liste
- `boolean contains(Object o)` retourne `true` si `o` existe dans la liste. Le test d'existence est réalisé en utilisant `equals` de `o`.

## Exercice 5 – Simulation de vélos en libre-service (20pts)

L'objectif de ce problème est de développer une simulation d'un système de location de vélos en libre-service (type Vélib). Nous nous concentrons sur la base du système de gestion des abonnements. Le simulateur doit pouvoir gérer :

- une flotte de vélos qui peuvent être électriques ou classiques ;
- différents types d'abonnements qui permettent de moduler la tarification selon le type de vélo et de garder la trace des locations d'une personne ;

Deux classes représentant les vélos sont nécessaires : la classe `Velo` modélisant un vélo classique et la classe `VeloElectrique` modélisant un vélo électrique. `Velo` contient un identifiant unique `String id` de la forme "C0", "C1", ... pour un vélo classique et de la forme "E0", "E1", ... pour un vélo électrique (les numéros sont indépendants pour chaque classe). Ces deux classes contiennent une méthode `void rendre(Abonnement a, int duree)` appelée lorsque le vélo est rendu et qui va permettre de mettre à jour toutes les informations de l'utilisateur.
Nous utiliserons un objet `Location` qui est un simple moyen de stockage d'un triplet d'informations `int duree, double prix, Velo v`. Cette classe très simple correspond au code suivant :

```
1 public class Location{
2   public final int duree; // en secondes
3   public final double prix;
4   public final Velo v;
5   public Location(int duree, double prix, Velo v){this.duree = duree; this.prix = prix; this.v = v;}
6   public String toString(){return "Duree␣:␣"+duree+"␣velo:␣"+v+ "␣prix:␣"+prix;}
7 }
```

**Q 5.1 (2pts)** Donner le code partiel de la classe `Velo` contenant deux constructeurs : un protected à un argument `String` mettant à jour `id` et un public sans argument. Vous ajouterez l(es) attribut(s) nécessaire(s) à la gestion de `id`. Le code de la méthode `rendre` sera demandé plus tard.

```
1 public class Velo  {
2   private String id;
3   private static int cpt=0;
4
5   public Velo(String id){
6           this.id = id;
7   }
8   public Velo(){id="V"+(cpt++);}
9
10  public void rendre(Abonnement a, int duree){
11
12  }
13 }
```

**Q 5.2 (1pt)** Un vélo électrique à en plus une variable `double charge` comprise entre 0 et 1, qui indique le niveau de la batterie et qui est initialisé à 1. Donner le code de la classe `VeloElec` qui ne contient qu'un constructeur sans argument.

```
1 public class VeloElec extends Velo {
2   private static int cpt;
3   private double charge ;
4   public VeloElec(){
5       super("E"+cpt++);
6       charge = 1.;
7   }
8 }
```

**Abonnements & tarification :** Le prix d'une location dépend à la fois du type de vélo emprunté (électrique ou classique) et du type d'abonnement. On distingue deux types d'abonnements : l'abonnement `AboLibre` qui facture 1 euro pour chaque location puis 0.01 euro par seconde pour vélo classique et 0.02 pour un vélo électrique et l'abonnement `AboMax` qui facture 0 euro pour chaque location et les 30 premières minutes sont gratuites quelque soit le vélo, puis 0.001 euro par seconde pour un vélo classique et 0.002 euro pour un vélo électrique. Ces deux classes sont les filles d'une même classe mère `Abonnement`. Vous empêcherez toute création d'instance de la classe `Abonnement`. Ces classes d'abonnement doivent permettre de conserver l'historique de toutes les locations d'un abonné. Ces classes contiennent un constructeur à un attribut et auront aussi une méthode `ajouter(Location loc)` et une méthode

`double getFacture()` qui permet de calculer la somme des factures de toutes les locations de l'abonnement. Nous allons aussi créer une méthode `String getFactureDetaillee()` qui donne les détails de chaque location.

**Q 5.3 (3pts)** Donner les codes de la classe mère `Abonnement` et d'une classe fille très sommaire `AboLibre` (pas besoin de donner `AboMax` pour l'instant). Il vous appartient de sélectionner la bonne structure de données pour stocker les `Location`.
Note : les calculs sur les prix de location sont abordés dans la question suivante.

```
1 public abstract class Abonnement {
2     private String name;
3     private ArrayList<Location> liste;
4
5     public Abonnement(String name) {
6         this.name = name;
7         liste = new ArrayList<Location>();
8     }
9
10    public void ajouter(Location loc) {
11        liste.add(loc);
12    }
13
14    public double getFacture() {
15        double sum = 0;
16        for(Location loc:liste)
17            sum += loc.prix;
18        return sum;
19    }
20
21    public String getFactureDetaillee() {
22        String ret = "";
23        for(Location loc:liste)
24            ret += loc + "\n";
25        return ret;
26    }
27 }
28
29 public class AboLibre extends Abonnement{
30
31    public AboLibre(String name) {
32        super(name);
33    }
34 }
```

**Q 5.4 (4pts)** Il reste maintenant à attaquer la partie difficile du problème : la méthode `public void rendre(Abonnement a, int duree)` dans les classes `Velo` et `VeloElec`. Cette méthode doit créer une instance de `Location` en calculant le prix associé à la location (qui dépend du type d'abonnement). Donner une première implémentation basée sur `instanceof`.

```
1     // dans Velo
2     public void rendre(Abonnement a, int duree){
3         double prix;
4         if(a instanceof AboMax) {
5             duree -= 30*60;
6             if(duree <= 0) prix = 0; // on peut le faire de manière élégante avec un Max
7             else prix = duree * 0.001;
8         }
9         else
10            prix = 1 + duree * 0.01;
11        a.ajouter(new Location(duree, prix, this));
12    }
13    // dans VeloElectrique
14    public void rendre(Abonnement a, int duree){
15        double prix;
16        if(a instanceof AboMax) {
17            duree -= 30*60;
18            if(duree <= 0) prix = 0; // on peut le faire de manière élégante avec un Max
19            else prix = duree * 0.002;
20        }
21        else
22            prix = 1 + duree * 0.02;
```

```
23          a.ajouter(new Location(duree, prix, this));
24      }
```

**Q 5.5 (3pts)** Donner le code de la classe `TestVelib` qui instancie deux abonnements (une de chaque type), deux vélos (un de chaque type). Vous allez créer un historique de 10 locations pour les deux abonnements en sélectionnant aléatoirement un vélo et une durée comprise entre 10x60 secondes et 40x60 secondes. Cet historique sera créé en exploitant la méthode `rendre` des vélos, qui sera implémenté dans la suite de cet énoncé.

```
1 public class TestVelib {
2
3     public static void main(String[] args) {
4         Velo v = new Velo();
5         VeloElec ve = new VeloElec();
6         AboMax amax = new AboMax("toto_max");
7         AboLibre ali = new AboLibre("titi_li");
8
9         for(int i=0; i<10; i++) {
10            // user 1
11            int duree = (int) (Math.random() * 30 * 60 + 10*60);
12            if(Math.random()<0.5) // une fois sur deux
13                v.rendre(amax, duree);
14            else
15                ve.rendre(amax, duree);
16
17            // user 2
18            duree = (int) (Math.random() * 30 * 60 + 10*60);
19            if(Math.random()<0.5) // une fois sur deux
20                v.rendre(ali, duree);
21            else
22                ve.rendre(ali, duree);
23
24        }
25        System.out.println(ali.getFactureDetaillee());
26        System.out.println(amax.getFactureDetaillee());
27
28    }
29 }
```

**Q 5.6 (2pts)** L'implémentation précédente de `rendre` n'est pas très satisfaisante, pourquoi ? [Deux raisons]

1. Elle n'est pas évolutive : on ne peut pas ajouter de nouvel abonnement sans revenir et modifier la classe Velo
2. Elle n'est pas raisonnable : on définit les paramètres de l'abonnement au niveau du velo

**Q 5.7 (3pts)** On propose d'ajouter dans tout `Abonnement` deux méthodes `double getCout(Velo v, int duree)` et `double getCout(VeloElec v, int duree)` qui vont calculer le coût d'une location en fonction de la nature du vélo et de l'abonnement. Donner le code à ajouter dans les classes `Abonnement` et dans les deux classes filles.

```
1 // dans Abonnement
2    public abstract double getCout(Velo v, int duree);
3    public abstract double getCout(VeloElec v, int duree);
4
5 // Dans AboMax
6     public double getCout(Velo v, int duree) {
7         duree -= 30*60;
8         if(duree <= 0) return 0;
9         return duree * 0.001;
10    }
11    public double getCout(VeloElec v, int duree) {
12        duree -= 30*60;
13        if(duree <= 0) return 0;
```

```
14          return duree * 0.002;
15      }
16
17  // Dans AboLibre
18      public double getCout(Velo v, int duree) {
19          return 1 + duree * 0.01;
20      }
21      public double getCout(VeloElec v, int duree) {
22          return 1 + duree * 0.02;
23      }
```

**Q 5.8 (2pts)** Donner une seconde version de l'implémentation de `public void rendre(Abonnement a, int duree)` qui exploite cette nouvelle architecture (uniquement pour la classe `Velo`).

```
1       public void rendre(Abonnement a, int duree){
2           double prix = a.getCout(this, duree);
3           a.ajouter(new Location(duree, prix, this));
4       }
```

**Q 5.9 (Bonus)** A votre avis, est-il nécessaire de redéfinir la méthode dans `Velo` et `VeloElec` ou seulement dans `Velo` ? Justifier succinctement.

Il faut mettre la méthode dans les deux classes.
Dans : `double prix = a.getCout(this, duree);;`
`this` désigne un Velo dans la classe Velo et on n'est pas dynamique sur les arguments des méthodes... Il faut donc mettre le même code dans la classe VeloElec pour que `this` désigne un VeloElec et que ça invoque la bonne méthode getCout !