# 7 Héritage et classe abstraite

*Remarque* : les frontières des TD à partir de la séance 7 sont flous. On peut taper dans les différentes séances et revenir en arrière sans problème
**Séance 7 :**
**Objectif :** classe/méthode abstraite
36, 37, (38), 39 (vraiment très dur mais très intéressant), 40 (rapide, c'est du cours)

## Exercice 36 – Figures (héritage, constructeurs, méthode abstraite)

Soit le programme Java constitué des classes suivantes :

```
1 public abstract class Shape {
2     protected double x, y ; // ancrage de la figure
3     public Shape() { x = 0 ; y = 0 ; }
4     public Shape(double x, double y) { this.x = x ; this.y = y ; }
5     public String toString() { return "Position : (" + x + "," + y + ")" ; }
6     public abstract double surface() ;
7 }
8
9 public class Circle extends Shape {
10     private double radius ;
11     public Circle() {
12       super();    // pas obligatoire (appel implicite) mais très recommandé
13       radius = 1 ;
14     }
15     public Circle(double x, double y, double r)
16     {
17       super(x,y) ;
18       radius = r ;
19     }
20     public String toString() {
21       return super.toString() + " Rayon : " + radius ;
22     }
23 }
24
25 public class MainShape {
26     public static void main(String [] args) {
27       Circle c1,c2 ;
28       c1 = new Circle(1,1,3) ;
29       c2 = new Circle() ;
30       System.out.println(c1.toString() + "\n" + c2.toString());
31     }
32 }
```

**Q 36.1** De quels membres (variables d'instance et méthodes) de `Shape` hérite la classe `Circle` ?

`Circle` hérite de x, y de la classe `Shape`,où il y a rédéfinition de la méthode `toString()`-héritée de la classe `Object`. La méthode `toString` est encore redéfinie dans la classe `Circle`. La méthode abstraite `surface()` de `Shape` devra être définie dans les sous-classes.

**Q 36.2** La compilation de la classe `Circle` échoue, expliquer pourquoi.

La méthode abstraite `surface()` n'est pas redéfinie dans la sous-classe `Circle`, donc soit la classe `Circle` devrait être déclarée abstraite, soit la méthode `surface()` devrait être redéfinie.
*Rappel* : Toute classe ayant une méthode abstraite est forcément abstraite.

**Q 36.3** Ajouter une méthode `surface()` à la classe `Circle` et modifier en conséquence la méthode `toString`.

```
1   public double surface() { return Math.PI*radius*radius; }
2   public String toString() {
3     return super.toString() + " Rayon : " + radius + " surface : " + surface();
4   }
```

**Q 36.4** Créer une classe `Rectangle` qui hérite de `Shape`.

```
1 public class Rectangle extends Shape {
2   private double h,l ;
3   public Rectangle() {
4     super();
5     l=1;h=1;
6   }
7   public Rectangle(double x, double y, double l,double h) {
8     super(x,y) ;
9     this.h=h ;
10    this.l=l;
11  }
12  public String toString() {
13    return super.toString() + " Rectangle de longueur " + l+ ", de hauteur "+ h + " et 
        de surface : " +surface();
14  }
15  public double surface() {
16    return l*h;
17  }
18 }
```

**Q 36.5** Donner le code d'un `main` qui instancie un tableau de `Shape`, le remplit avec différents types de forme puis calcule l'aire totale de la figure composite (sans prendre en compte les recouvrements).

```
1 Shape[] tab = new Shape[3];
2 tab[0] = new Circle();
3 tab[1] = new Rectangle();
4 tab[2] = new Rectangle(1, 1, 3, 5);
5 double aire = 0;
6 for(Shape s:tab) aire += s.surface();
```

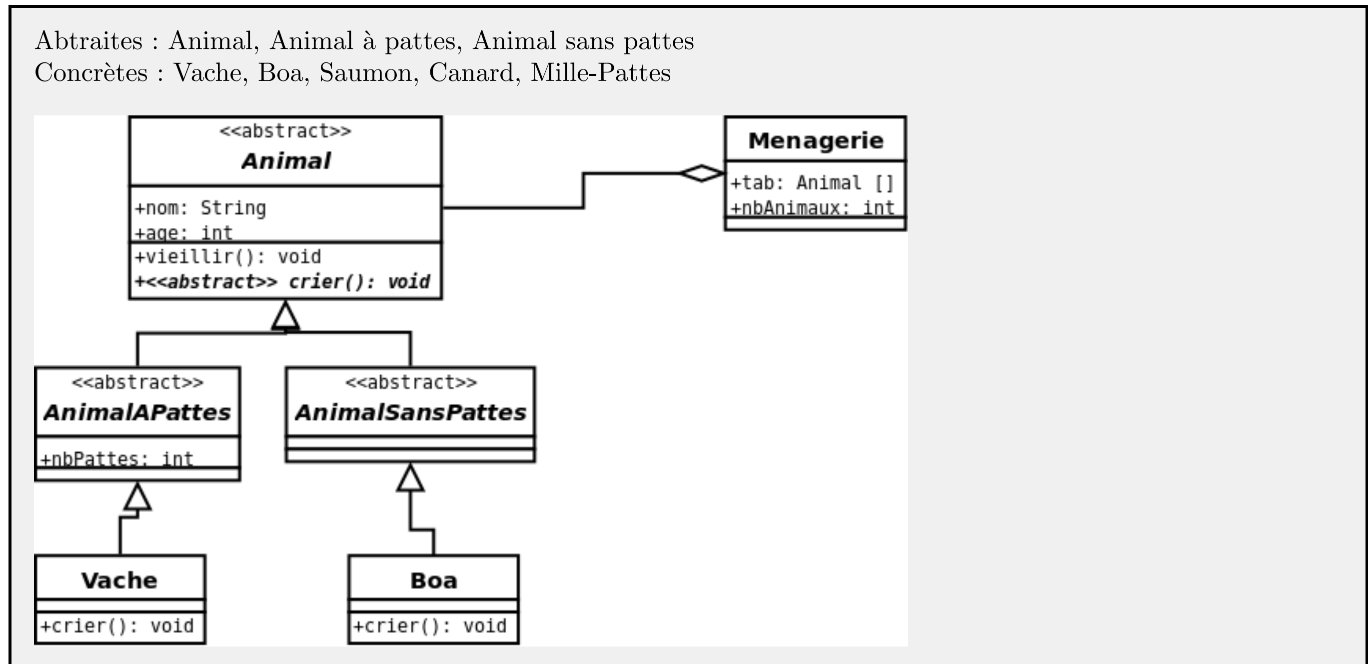
## Exercice 37 – Ménagerie (tableaux, héritage, constructeur)

Exercice sur redéfinition de méthode et classe `abstraite`

On veut gérer une ménagerie dont les animaux ont chacun un nom (String) et un âge (int). Parmi ceux-ci on distingue les animaux à pattes (variable nbPattes) et les animaux sans pattes. On s'intéresse uniquement aux vaches, boas, saumons, canards et mille-pattes.

**Q 37.1** Etablir graphiquement la hiérarchie des classes ci-dessus. Déterminer celles qui peuvent être déclarées abstraites ?

Abtraites : Animal, Animal à pattes, Animal sans pattes
Concrètes : Vache, Boa, Saumon, Canard, Mille-Pattes




**Q 37.2** Ecrire la classe `Animal` avec deux constructeurs (un prenant en paramètre le nom et l'âge, l'autre prenant en paramètre le nom et qui fixe l'âge à 1 an), la méthode `toString`, une méthode vieillir qui fait vieillir l'animal d'une année, et une méthode `crier()` qui affichera le cri de l'animal. Peut-on écrire ici le corps de cette méthode ?

// Cette classe `est` abstraite car elle contient une methode abstraite

```
1  public abstract class Animal {
2    private String nom;
3    private int age;
4    public Animal(String nom, int age) {
5      this.nom=nom;
6      this.age=age;
7    }
8    public Animal(String nom) {
9      this(nom,1);
10   }
11   public void vieillir() {
12     age++;
13   }
14   public abstract void crier(); /** Methode abstraite */
15   public String toString() {
16     return "Je m'appelle "+nom+", j'ai "+age +"ans";
17   }
18 }
```

**Q 37.3** Ecrire toutes les sous-classes de la classe `Animal` en définissant les méthodes `toString()` et les méthodes `crier()` qui affichent le cri de l'animal.

```
1 // Cette classe est abstraite, car elle herite de la methode abstraite crier()
2
3   public abstract class AnimalAPattes extends Animal {
4     private int nbPattes;
5     public AnimalAPattes(String nom, int n) {
6       super(nom);
7       nbPattes=n;
8     }
9     public String toString() {
10      return super.toString()+", j'ai "+nbPattes+" pattes";
11    }
12  }
13
14 // Cette classe est abstraite, car elle herite de la methode abstraite crier()
15  public abstract class AnimalSansPattes extends Animal{
16    public AnimalSansPattes(String nom){
17      super(nom);
18    }
19  }
20  public class Vache extends AnimalAPattes {
21    public Vache(String nom) { super(nom,4); }
22    /** On donne ici le corps de la methode crier()*/
23    public void crier() { System.out.println(" Meuuuh ! "); }
24    public String toString() {
25      return super.toString()+" et je suis une vache ";
26    }
27  }
28  public class Boa extends AnimalSansPattes {
29    public Boa(String nom) {
30      super(nom);
31    }
32    public void crier(){
33      System.out.println(" SSSSSSSS! ");
34    }
35    public String toString() {
36      return super.toString()+" et je suis un boa";
37    }
38  }
```

De meme, pour Saumon et Canard

**Q 37.4** Ecrire une classe `Menagerie` qui gère un tableau d'animaux, avec la méthode `void ajouter(Animal a)` qui ajoute un animal au tableau, et la méthode `toString()` qui rend la liste des animaux.

```
1 public class Menagerie {
2     private Animal [] tab;
3     private int nbAnimaux=0;
4     public Menagerie(int taille) {
5         tab=new Animal[taille];
6     }
7     public void ajouter(Animal a) {
8         if (nbAnimaux==tab.length) {
9             System.out.println("La menagerie est pleine !\n");
```

```
10              return; // Fin de la methode
11          }
12          tab[nbAnimaux]=a;
13          nbAnimaux++;
14      }
15      public Animal enlever()} {
16          if (nbAnimaux==0) {
17              System.out.println("La menagerie est vide !\n");
18              return null; // Fin de la methode
19          }
20          nbAnimaux--;
21          Animal a=tab[nbAnimaux];
22          tab[nbAnimaux]=null;
23          return a;
24      }
25      public String toString() {
26          String s="";
27          for (int i=0; i<nbAnimaux; i++)
28              s += tab[i]+"\n";
29          return s;
30      }
31  }
```

**Q 37.5** Ajouter une méthode `void midi()` qui fait crier tous les animaux de cette ménagerie.

**Q 37.6** Ecrire la méthode `vieillirTous()` qui fait vieillir d'un an tous les animaux de cette ménagerie.

```
1   /** Comme tous les animaux contiennent la methode crier,
2   * on peut parcourir le tableau sans faire attention au type de l'animal. */
3
4   public void midi() {
5       for (int i=0; i<nbAnimaux; i++) {
6           tab[i].crier();
7       }
8   }
9   public void vieillirTous() {
10      for (int i=0; i<nbAnimaux; i++)
11          tab[i].vieillir();
12  }
```

**Q 37.7** Ecrire la méthode `main` qui crée une ménagerie, la remplit d'animaux, les affiche avec leur âge, déclenche la méthode `midi()` et les fait vieillir d'un an.

Correction du main :

```
1   public class TestMenagerie {
2     public static void main(String [] args) {
3       Menagerie m=new Menagerie(20);
4       Animal b1= new Boa("Beatrice");
5       Animal b2= new Boa("Bernard");
6       Animal v1= new Vache("Marguerite");
7       Animal v2= new Vache("Blanchette");
8       m.ajouter(b1);
9       m.ajouter(b2);
```

```
10      m.ajouter(v1);
11      m.ajouter(v2);
12      System.out.println("Il est midi :");
13      m.midi();
14      System.out.println(m);
15      m.vieillirTous();
16      System.out.println(m);
17    }
18  }

---- EXECUTION ----
Il est midi :
SSSSSSSS!
SSSSSSSS!
Meuuuh !
Meuuuh !
Je m'appelle Beatrice, j'ai 3 ans et je suis un boa
Je m'appelle Bernard, j'ai 3 ans et je suis un boa
Je m'appelle Marguerite, j'ai 3 ans, j'ai 4 pattes et je suis une vache
Je m'appelle Blanchette, j'ai 3 ans, j'ai 4 pattes et je suis une vache
Je m'appelle Beatrice, j'ai 4 ans et je suis un boa
Je m'appelle Bernard, j'ai 4 ans et je suis un boa
Je m'appelle Marguerite, j'ai 4 ans, j'ai 4 pattes et je suis une vache
Je m'appelle Blanchette, j'ai 4 ans, j'ai 4 pattes et je suis une vache
```

## Exercice 38 – Figure2D (Extrait de l'examen de janvier 2010)

On veut écrire les classes correspondant à la hiérarchie suivante (le niveau d'indentation correspond au niveau de la hiérarchie) :

```
Figure (classe abstraite)
    |____Figure2D (classe abstraite)
             |____Rectangle
                     |____Carre
             |____Ellipse
                     |___Cercle
```

Ces classes devront respecter les principes suivants :

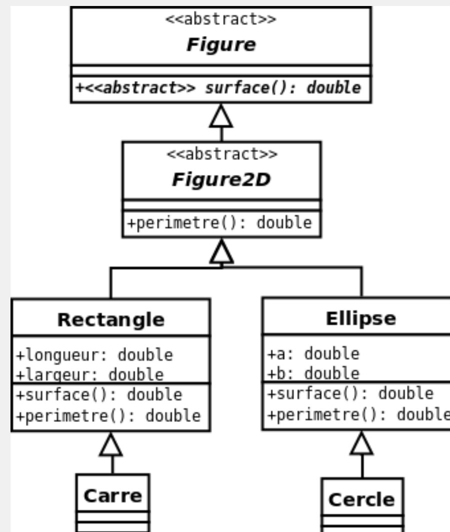
- Toutes les variables d'instance sont de type `double` et caractérisent uniquement la taille des objets, pas leur position.
- Chaque objet sera créé par un constructeur qui recevra les paramètres nécessaires (par exemple la longueur et la largeur d'un rectangle).
- Toutes les instances devront accepter les méthodes `surface()` et `toString()`.
- Toutes les instances d'objets de type 2D devront accepter la méthode `perimetre()`.

*Rappel sur les ellipses :* une ellipse est caractérisée par la longeur a du demi-grand axe et la longueur b du demi-petit axe. Sa surface est $\pi * a * b$ et son périmètre est $2\pi\sqrt{\frac{(a^2+b^2)}{2}}$.

*Rappel :* dans la classe `Math`, il existe la constante `Math.PI` et la méthode `Math.sqrt()` qui retourne la racine carrée d'un nombre (voir annexe page 135).

**Q 38.1** Quelles sont les particularités d'une méthode abstraite et les conséquences pour la classe et les classes dérivées ?

Une méthode abstraite est une méthode dont on ne précise que l'en-tête (signature) et qui sera implémenter dans les classes concrètes.
Toute classe `contenant` une méthode abstraite est forcément abstraite.
Une classe `concrète` se doit d'implémenter les méthodes abstraites de la super-classe.




**Q 38.2** Donner pour chacune des classes, en utilisant correctement les notions d'héritage et de classe abstraite :
- la définition de la classe,
- la déclaration des variables d'instance,
- le constructeur,
- les méthodes de la classe.

```java
1 public abstract class Figure {
2     public abstract double surface();
3     public String toString() { return "c'est une figure";}
4 }
5 public  abstract class Figure2D extends Figure {
6     public abstract double perimetre();
7 }
8 public  class Rectangle extends Figure2D {
9     private double longueur, largeur;
10    public Rectangle (double l1, double l2) {
11      super();
12      longueur = l1;
13      largeur = l2;
14    }
15    public String toString() {
16      return "c'est un rectangle";
17    }
18    public double surface() {
19      return longueur*largeur;
20    }
21    public double perimetre() {
22      return 2*(longueur+largeur) ;
23    }
24 }
25 public class Carre extends Rectangle {
```

```
26      Carre(double cote) {
27        super(cote, cote);
28      }
29 }
30 public class Ellipse extends Figure2D {
31      private double a, b;
32      public Ellipse (double a, double b) {
33        super();
34        this.a = a;
35        this.b = b;
36      }
37      public String toString() {
38        return "c'est une ellipse";
39      }
40      public double surface() {
41        return Math.PI*a*b;
42      }
43      public double perimetre() {
44        return 2*Math.PI*Math.sqrt(a*a+b*b)/2;
45      }
46    }
47 public class Cercle extends Ellipse {
48      Cercle(double rayon) {
49        super(rayon, rayon);
50      }
51    }
```

NB : l'héritage entre le cercle et l'ellipse constitue un débat sans fin... Il est intéressant d'expliquer les deux solutions et de justifier l'usage de cette correction : le cercle étend ellipse car un Cercle EST UNE Ellipse du point de vue géométrique (vérité terrain). Dans l'idéal, on choisit toujours de faire l'héritage qui correspond à la vérité terrain.

**Q 38.3** Écrire une classe appelée `TestFigure` qui contient une méthode `main`. Cette méthode créera un objet de chacun des types, et affichera sa surface et son périmètre.

```
1 public  class TestFigure {
2      public static void main (String [] args) {
3        Rectangle r = new Rectangle( 10,4);
4        System.out.println (r.toString()+" dont la surface est : "+ r.surface() + " et le
             perimetre : "+ r.perimetre());
5        Carre c = new Carre(25);
6        System.out.println (c.toString()+" dont la surface est : "+ c.surface() + " et le
             perimetre : "+ c.perimetre());
7        Ellipse e = new Ellipse(24, 12);
8        System.out.println (e.toString()+" dont la surface est : "+ e.surface() + " et le
             perimetre : "+ e.perimetre());
9        Cercle cc = new Cercle(15);
10       System.out.println (r.toString()+" dont la surface est : "+ r.surface() + " et le
             perimetre : "+ r.perimetre());
11     }
12 }
```

## Exercice 39 – Retro engineering

Soit le programme principal suivant permettant d'effectuer des opérations mathématiques très simples dans un nouvel univers objet. Comme le précise le `main` suivant, une expression est soit une valeur réelle, soit une opération mathématique. Pour ne pas complexifier la situation, nous n'envisageons que des opérations réelles (sur des `double`).

```
1 public static void main(String args[]){
2         Expression v1=new Valeur(4.);
3         Expression v2=new Valeur(1.);
4         Expression v3=new Valeur(7.);
5         Expression v4=new Valeur(5.);
6         Expression v5=new Valeur(3.);
7         Expression v6=v5;
8         Operation p1=new Plus(v1,v2);
9         Operation m2=new Moins(v3,v4);
10        Operation mult=new Multiplie(p1,v5);
11        Operation p2=new Plus(v6,mult);
12        Operation d=new Divise(p2,m2);
13        System.out.println(d+"="+d.getVal());
14 }
```

**Q 39.1** Donner la hiérarchie des classes (avec les **signatures** de méthodes abstraites et concrètes et la signature du constructeur lorsqu'il est nécessaire) à définir pour que ce programme puisse compiler et s'exécuter.
**Attention :** on veut que la dernière ligne du `main` affiche le calcul à effectuer dans le détail (cf question suivante)

```
1 Expression (ABS)
2  + ABS double getVal()
3         -> Valeur
4             + double getVal()
5             +String toString()
6         -> Operation (ABS)
7                 -> Plus
8                   + Plus(Expression, Expression)
9                   + double getVal()
10                  +String toString()
11                -> Moins
12                  + Moins(Expression, Expression)
13                  + double getVal()
14                  +String toString()
15                -> Multiplie
16                  + Multiplie(Expression, Expression)
17                  + double getVal()
18                  +String toString()
19                -> Divise
20                  + Divise(Expression, Expression)
21                  + double getVal()
22                  +String toString()
```

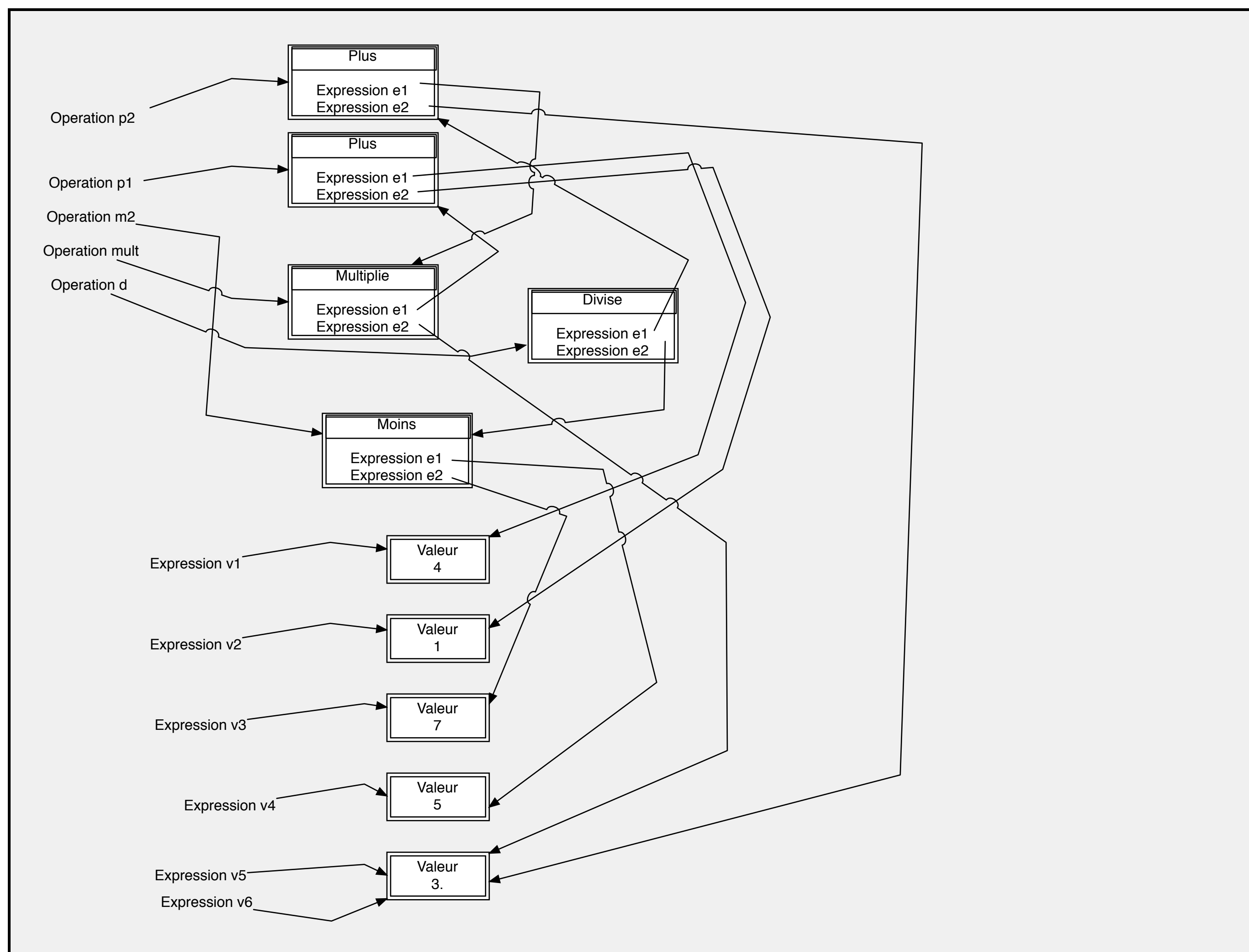
**Q 39.2** La hiérarchie de classes proposée définit une expression arithmétique qui peut être évaluée pour donner un résultat (méthode `getVal()`). Donner l'expression arithmétique (avec parenthèses) correspondant à l'objet `d` du programme donné ci-dessus.

$$(((4 + 1) * 3) + 3)/(7 - 5)$$

**Q 39.3** Donner le code des classes nécessaires pour que le programme s'exécute et affiche la formule évaluée et son résultat en ligne 13 (le code des classes Plus, Moins, Multiplie et Divise étant très proche, on ne donnera le code que de Divise).

```
1 public abstract class Expression {
2     public abstract double getVal();
3 }
4
5 public class Valeur extends Expression{
6     private double val;
7     public Valeur(double val) {
8         super(); //OPT
9         this.val = val;
10     }
11
12     public double getVal() {
13         return val;
14     }
15      public String toString(){
16         return ""+val; // 0.25 de penalite pour l'oubli de ""+
17     }
18 }
19
20 public abstract class Operation extends Expression{
21 }
22
23 public class Plus extends Operation{
24     private Expression e1, e2;
25
26     public Plus(Expression e1, Expression e2) {
27         super();
28         this.e1 = e1;
29         this.e2 = e2;
30     }
31
32     public double getVal() {
33             return e1.getVal()+e2.getVal();
34     }
35     public String toString(){
36         return "("+e1.toString()+"+"+e2.toString()+")";
37     }
38 }
```

**Q 39.4** Donner le diagramme de l'état de la mémoire à la fin du programme (ligne 13).




**Q 39.5** On souhaite maintenant pouvoir modifier l'attribut d'un objet `Valeur`. On ajoute alors la fonction `void setVal(double v)` à la classe `Valeur` qui fixe à `v` l'attribut de la classe. Soit la ligne de code suivante :

```
1 v6.setValeur(4);
```

En l'état, le programme ne compile pas. Pourquoi ? Donner deux manières de remédier au problème. Discuter brièvement des avantages / inconvénients de ces deux manières de faire.

setValeur pas définie pour Expression
- ((Valeur)v6).setValeur(5)
- ou déclarer une fonction abstraite setValeur dans la classe Expression (mais çà implique de la définir partout en dessous donc pas terrible)

**Q 39.6** Quel problème survient dans l'exécution du programme si l'on remplace la ligne 5 par :

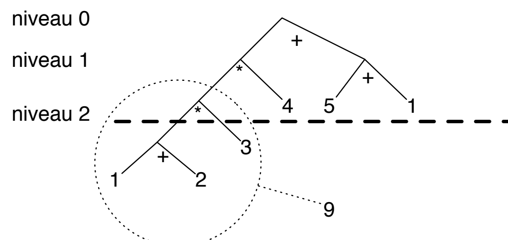
```
1 Expression v4=new Valeur(7);
```

Division par 0

**Q 39.7** Dire comment y remédier pour que le programme affiche le message d'erreur approprié et évite un arrêt brutal du programme. Décrire brièvement les méthodes à modifier et les éventuelles classes à créer.

> ATTENTION : copier coller d'examen. Si vous n'en êtes pas là en TD, on saute
> (1) Définir une classe Nan qui hérite de Expression et qui se propage dans toutes les opérations
> (2) Ajouter un try/catch au niveau des getVal qui retourne Nan + un message en cas de problème

**Q 39.8** En voyant une expression comme un arbre, on souhaite développer une méthode `simplifie(int profondeur)` permettant la simplification d'une expression à partir d'une profondeur donnée, avec `profondeur` un entier supérieur à 0. Lorsque la profondeur désirée est atteinte, cette simplification consiste à remplacer l'expression concernée par un objet Valeur de valeur équivalente. Par exemple, l'objet Expression dont la formule est $((((1+2)*3)*4)+(5+1))$ se simplifie en $((9*4)+(5+1))$ par l'appel de `simplifie(2)`. Donner le code permettant cette simplification.




```
// Expression
public abstract Expression simplifie(int level);

// Valeur
public Expression simplifie(int level){
  return this;
}

// Plus
public Expression simplifie(int level){
  if(level >0)
    return new Plus(e1.simplifie(level-1), e2.simplifie(level-1));
  return new Valeur(getVal());
}
```

## Exercice 40 – Final : les différentes utilisations

**Q 40.1** Questions de cours

**Q 40.1.1** A quoi sert un attribut `final` ? Où peut-il être initialisé ? Citer des cas d'utilisation.

Un attribut dont la valeur ne peut jamais être changée

```
// Usage 1
public class Truc{
  private final int i = 1;
// Usage 2
public class Truc{
  private final int i;
  public Truc(){
    i = 1; // init possible dans le constructeur
  }
```

Pour les constantes (comme $\pi$...) : pour être sur que la valeur ne change pas
Pour les choses qui doivent être fixées une fois pour toutes (identifiants...)

**Q 40.1.2** Dans quel cas déclarer une méthode comme `final` ?

si on ne veut pas qu'elle soit redéfinie dans une classe fille : on estime que le comportement ne peut pas changer dans le futur
cf exemple dans la question suivante
NB : apparemment d'après le web, aucun gain de performance à attendre...

**Q 40.1.3** Dans quel cas déclarer une classe comme `final` ?

Si on ne veut pas qu'elle soit redéfinie : une classe pour une tache simple
Exemple : Integer, Double...

**Q 40.1.4** Etant donné les usages répertoriés ci-dessus, à quoi sert le mot clé `final` en général ?

A améliorer la sécurité du programme en évitant des commandes "interdites"

**Q 40.2** Application sur la classe Point

```
1 public class Point {
2     private double x,y;
3     private static int cpt = 0;
4     private int id;
5
6     public Point(double x, double y) {
7         this.x = x;  this.y = y; id = cpt ++;
8     }
9     public double getX() { return x;}
10    public double getY() { return y;}
11    public String toString() {
12        return "Point␣[x=" + x + ",␣y=" + y + "]";
13    }
14    public void move(double dx, double dy){ x+=dx; y+=dy;}
```

**Q 40.2.1** Au niveau des attributs, serait-il intéressant d'ajouter le modifier `final` sur certains champs ? Pourquoi ?

**Q 40.2.2** A quelle condition pourrait-on mettre `x` et `y` en mode `final` ? Proposer une solution pour conserver les fonctionnalités de la classe.

— sur `id` : une fois l'identifier régler, il ne change plus... L'init est faite dans le constructeur, c'est OK.

```
1 private final int id;
```

— sur x,y : incompatible avec la méthode move, on ne peut pas passer en final.
S'il n'y avait pas la méthode `move`, on pourrait mettre `x,y` en final... On aurait alors un comportement à la String : pour modifier il faut créer une nouvelle instance. Il n'y a plus aucun problème de clonage ou de références... C'est très sécurisé

```
1 public class Point {
2     private final double x,y;
3
4     public Point createTranslatedPoint(double dx, double dy){
5       return new Point(x+dx, y+=dy);
6     }
7 }
```

Une telle modification est intéressante : si on considère l'addition de deux Point, il y a toujours une ambiguïté pour savoir si l'addition modifie le point courant ou génère une nouvelle instance... Avec final, plus de doute !

**Q 40.2.3** Quelles fonctions pourraient être `final` ? Quel serait l'intérêt de la manipulation ?

> Toutes les fonctions pourraient être final... sauf toString(). Les méthodes sont très simples, on garantit la fonctionnalité pour toutes les classes filles en déclarant les méthodes final : il est impossible qu'à un niveau donné le comportement de ces méthodes soit changé. Encore une fois : sécurisation

**Q 40.2.4** Quel serait l'intérêt de déclarer la classe `final` ? Cela empêche-t-il tout enrichissement futur ?

**Q 40.2.5** Proposer un code pour la classe `PointNomme` (point ayant un attribut `nom`) après avoir déclaré Point en `final`.

> Classe final = impossible de créer une classe fille... Mais il est possible d'enrichir une classe par composition/délégation

```java
public class PointNomme{
  private Point p;
  private String nom;
  public PointNomme(double x, double y, String nom){
    this.nom = nom; p = new Point(x,y);
  }

  public double getX(){return p.getX();} // delegation
```

> Avec ce système, une chose fondamentale change : un PointNomme N'EST PAS un Point... Si une méthode est définie comme : `void maMethode(Point p)`, on ne peut pas lui donner un PointNomme...
> Securisation (mais aussi limitation) de l'environnement de la classe déclarée final.

## Quizz 10 – Classe et méthode abstraite

**QZ 10.1** Les instructions suivantes sont-elles correctes ? Expliquez.

```java
public abstract class Z {}
public class TestQuizzAbstract {
        public static void main(String [] args) {
                Z z=new Z();
        }
}
```

> Rappel de cours : Si une classe est déclarée `abstraite` alors on ne peut pas créer d'objet de cette classe (pas de `new ClasseAbstraite()`).
> Z est une classe abstraite. On ne peut pas créer d'instance de cette classe.
> TestQuizzAbstract.java :4 : Z is abstract ; cannot be instantiated
> Remarque : une classe abstraite ne possède pas forcemment une méthode abstraite.

**QZ 10.2** Les instructions suivantes sont-elles correctes ? Expliquez chaque erreur.

```java
public class Z {
        public abstract void f();
        public abstract void g() { } ;
        public void h();
}
```

Rappel de cours : une méthode abstraite est une méthode sans corps, elle n'a qu'une en-tête. Si une classe possède une méthode abstraite, cette classe doit être déclarée abstraite.
f() est une méthode abstraite, donc la classe Z doit être déclarée abstraite.
g() est une méthode abstraite, donc elle ne doit pas contenir de corps entre accolades ou bien elle ne doit pas contenir abstract, c'est ou l'un ou l'autre.
h() ne possède pas de corps (pas d'accolades), elle doit donc soit être déclarée abstraite, soit avoir des accolades.

```
1 public abstract class Z {
2         public abstract void f();
3         public abstract void g(); // ou public void g() { }
4         public abstract void h(); // ou public void h() { }
5 }
```

**QZ 10.3** Les instructions suivantes sont-elles correctes ? Expliquez et proposez deux solutions.

```
1 public abstract class A {
2         public abstract void f();
3 }
4 public class B extends A {}
```

Rappel de cours : Si une classe hérite d'une méthode abstraite, elle doit soit être déclarée abstraite, soit définir le corps de la méthode abstraite.
B hérite d'une classe abstraite, il faut soit la déclarée abstraite, soit définir le corps de la méthode abstraite.
Solution 1 : on déclare B abstraite.

```
1
2 public abstract class B extends A {}
```

Solution 2 : on définit le corps de la méthode f()

```
1 public class B extends A {
2         public void f() {}
3 }
```

## Quizz 11 – Vocabulaire sur l'héritage

En utilisant quelques verbes de l'ensemble ci-après, écrire trois courtes phrases caractérisant l'héritage : implémenter, instancier, importer, réemployer, ajouter, encapsuler, étendre, spécifier, redéfinir.
L'héritage permet de : ...

Quelques phrases possibles :
L'héritage permet de réemployer les champs et méthodes d'une classe ancêtre
L'héritage permet d'ajouter des éléments spécifiques, champs et/ou méthodes,
L'héritage permet de redéfinir le comportement de certaines méthodes.