1 Résolution d'une équation du second degré - Polymorphisme

Réalisez un programme Java/Lisaac calculant la résolution d'une équation du second degré de la forme $ax^2+bx+c=0$. Pour cela, vous prendrez soin de produire 3 classes définissant les 3 cas possibles (0 solution, 1 solution et 2 solutions). C'est 3 classes utiliseront l'héritage, avec comme parent commun une classe de type Binome donnée comme suit :

```
public class Binome {
  // Donnees:
  protected double a,b,c,dis;

  // Methodes
  public Binome(double pa,double pb,double pc,double pdis)
  { a = pa; b = pb; c = pc; dis = pdis; }

  public static Binome creation(double pa,double pb,double pc)
  { double delta;
    delta = pb * pb - 4.0 * pa * pc;
    if (delta < 0.0) {
      return new BinomeSol0(pa,pb,pc,delta);
    } else if (delta == 0.0) {
      return new BinomeSol1(pa,pb,pc,delta);
    } else {
      return new BinomeSol2(pa,pb,pc,delta);
    };
  }

  public void calculer_racine()
  // Mettre le/les résultat(s) dans des attributs de l'objet
  { System.out.println("Erreur si ici !"); }

  public int nb_racine()
  // Renvoi 0, 1 ou 2 (à definir dans les 3 classes feuilles)
  { System.out.println("Erreur si ici !"); return 0; }

  public double valeur_racine(int i)
  // Renvoi ième racine (déjà calculé par calculer_racine()
  { System.out.println("Erreur si ici !"); return 0.0; }

  public static int main(String arg[])
  { Binome b[3];
    int i,k;
    b[0] = Binome.creation (1.0,0.0,1.0);
    b[1] = Binome.creation (1.0,2.0,1.0);
    b[2] = Binome.creation (1.0,3.0,1.0);
    for (i=0;i<3;i++) {
      System.out.print("#"+i+": ");
      b[i].calculer_racine();
      if (b[i].nb_racine() == 0) {
        System.out.println("Pas de solution!");
```

```
      } else {
        for (k=1;k<=b[i].nb_racine();k++) {
          System.out.println(""+b[i].valeur_racine(k)+" ");
        };
      };
    };
    return 0;
  };
}
```

En Lisaac (dans une fichier `binome.li`) :

```
Section Header
  + name := BINOME;

Section Inherit
  - parent_object:OBJECT := OBJECT;

Section BINOME
  // Donnees:
  + a:REAL_32;
  + b:REAL_32;
  + c:REAL_32;
  + dis:REAL_32;

Section Public
  // Methodes:
  - create (pa,pb,pc,pdis:REAL_32) :SELF <-
  ( (a,b,c,dis) := (pa,pb,pc,pdis);
    clone
  );

  - creation (pa,pb,pc:REAL_32) :BINOME <-
  ( + delta:REAL_32;
    + result:BINOME;
    delta := pb * pb - 4.0 * pa * pc;
    (delta < 0.0).if {
      result := BINOME_SOL0;
    }.elseif {delta = 0.0} then {
      result := BINOME_SOL1;
    } else {
      result := BINOME_SOL2;
    };
    result.create (pa,pb,pc,delta)
  );

  - calculer_racine <-
  // Mettre le/les résultat(s) dans des attributs de l'objet
  ( deferred; );

  - nb_racine:INTEGER <-
  // Renvoi 0, 1 ou 2 (à definir dans les 3 classes feuilles)
  ( deferred; 0)
  [ ? Result.in_range 0 to 2; ]; // Post-condition !!
```

```
  - valeur_racine i:INTEGER :REAL_32 <-
  [ ? {i.in_range 1 to (nb_arcine)}; ] // Pré-condition !!
  // Renvoi ième racine (déjà calculé par calculer_racine()
  ( deferred; 0.0);

  - main <-
  ( + b:ARRAY BINOME;
    b := ARRAY BINOME.create_with_capacity 3;
    b.add_last (creation (1.0,0.0,1.0));
    b.add_last (creation (1.0,2.0,1.0));
    b.add_last (creation (1.0,3.0,1.0));
    0.to 2 do { i:INTEGER;
      '#'.print; i.print; ": ".print;
      b.item i.calculer_racine;
      (b.item i.nb_racine = 0).if {
        "Pas de solution!".println;
      } else {
        1.to (b.item i.nb_racine) do { k:INTEGER;
          b.item i.valeur_racine k .print; " ".println;
        };
      };
    };
  );
```

Dans un fichier `binome_sol0` :

```
Section Header
  + name := BINOME_SOL0;

Section Inherit
  + parent_binome:Expanded BINOME;

Section Public

  - calculer_racine;  // Methode vide

  - nb_racine:INTEGER; // valeur par défault 0

  // l'appel éventuel de valeur_racine va faire péter la pré-condition !!!
```

2 Animaux - Utilisation de super

La classe `Animal` comporte deux sous-classes `Mammifère` et `Poisson` ; la classe `Mammifère` comporte elle-même deux nouvelles sous-classes : `Chien` et `Homme`. Modéliser et créer ces différentes classes en Java/Lisaac de telle sorte que la classe `TestAnimal` donnée comme suit :

```
public class TestAnimal {
  public static void main(String[] args) {
    Animal[] animaux = new Animal[5];
    animaux[0]=new Animal("Truc");;
    animaux[1]=new Animal();
    animaux[2]=new Chien("Medor");
    animaux[3]=new Homme() ;
```

```
    animaux[4]=new Homme ("Robert") ;
    for (int i=0; i<5; i++) {
      System.out.println(animaux[i].getType());
    }
  }
}
```

L'affiche attendu est le résultat suivant :

```
Je suis un animal de nom Truc.
Je suis un animal.
Je suis un animal de nom Medor. Je suis un mammifere. Je suis un chien.
Je suis un animal. Je suis un mammifere. Je suis un homme.
Je suis un animal de nom Robert. Je suis un mammifere. Je suis un homme.
```

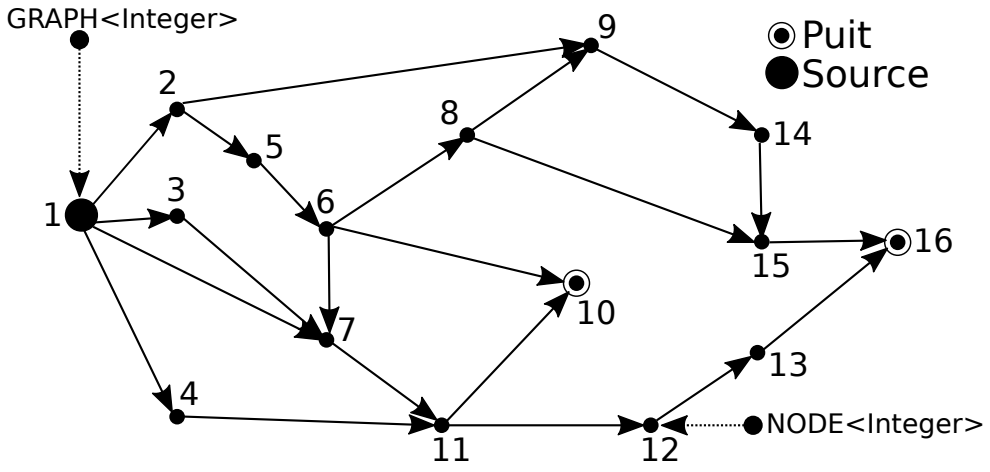
3 Graphe orienté non cyclique - Généricité

Nous proposons d'analyser la topologie et de fournir une librairie concernant les graphes orientés non cycliques.

Notre graphe a les propriétés suivantes :

— Il possède une seules sources. Cela représente les points d'entrées du graphe. En d'autre terme, aucun nœud pointe sur les nœuds sources.
— Il possède un ou plusieurs puits. Cela représente les points de destination du graphe. En d'autre terme, ces nœuds pointent sur aucun autre nœud.
— Le graphe est composé de différents nœuds inter-connectés par des arcs.
— Chaque arc est orienté.
— Nous simplifions notre graphe, en interdisant un cycle de nœuds.
— Chaque nœud possède une valeur représentée par un objet de type générique.

Voici un exemple :



Questions :

1. Programmez une classe NODE<E> permettant de modéliser un nœud de notre graphe.
2. Programmez une classe GRAPH<E> permettant de modéliser un graphe.
3. Programmez une méthode capable de donner le nombre minimum de nœuds qu'il est nécessaire de parcourir pour atteindre un puit partant de la source (appel principal sur un GRAPH).
4. Programmez une méthode capable de donner le nombre maximum de nœuds qu'il est nécessaire de parcourir pour atteindre un puit partant d'une source.
5. Programmez une procédure permettant de parcourir et d'afficher chaque nœud en profondeur d'abord. Dans l'exemple, ce type de parcours donne : 1, 2, 9, 14, 15, 16, 5, 6, 8, 10, 7, 11, 12, 13, 3, 4.

6. Puis un autre capable de parcourir chaque nœud en largeur d'abord. Dans l'exemple, ce type de parcours donne : (1), (2, 3, 7, 4), (9, 5, 11), (14, 6, 10, 12), (15, 8, 13), (16).
7. Concevez un itérateur (*implements Iterator*) à l'aide d'une classe emboitée permettant de parcourir chaque nœud en profondeur d'abord. Vous pouvez tester avec un `foreach` :

```
for (Integer i:my_graph) System.out.println("N:"+i);
```

Remarque : Ce type de graphe peut représenter un certain type de labyrinthe, un parcours de marchandise à optimiser, un arbre d'héritage multiple, ...