TP de Java
Classes et Objets

Sujets abordés dans ce TP :

Introduction P.O.O en Java
Premières notions d'héritage
Règles d'écriture des données membres et des méthodes
Communication entre classes
Utilisation de package
Algorithmes de parcours d'arbres binaires

1) Introduction

Dans ce TP nous vous proposons d'approfondir la P.O.O (Programmation Orientée Objets) en Java. On rappelle que la programmation Java est basée sur la manipulation d'objets composés de données membres et de méthodes. Les objets sont des instances de classes. La classe correspond à la généralisation de type, c'est une description d'un ensemble d'objets ayant une structure de données commune et disposant des mêmes méthodes.

2) Introduction P.O.O en Java

Déclaration de classes, allocation d'objets, encapsulation de données membres

Implémenter la classe suivante :

```
class Point {
        private int x,y;char l;

        private void check()
                {if(x<0) x=0; if(y<0) y=0;}
        void move(int dx, int dy)
                {x+=dx; y+=dy; check();}
        void set(int xi,int yi)
                {x = xi; y= yi; check();}

        int getX() {return x;}
        int getY() {return y;}

        void print()
                {System.out.println(x+";"+y+":"+l);}
        }
```

Mettre en oeuvre cette classe de la façon suivante:

```
Point pt = new Point();
pt.print();
pt.l = 'o';
pt.set(5,2);
pt.print();
pt.move(-6,4);
pt.print();
```

A travers ce court exemple vous avez vu les principes de base de la P.P.O avec : la classe *Point*, ses différentes méthodes, ses données membres *x* *y* et *l* (un label), et l'objet *pt* instance de la classe *Point*, alloué suite à l'appel de l'opérateur *new*.

Les méthodes et données membres des objets sont accessibles par l'utilisation de l'opérateur point ., comme par exemple *pt.print()* et *pt.l*.

Cet exemple met en œuvre un principe important de la P.O.O. : l'encapsulation des données membres. En effet, les données membres *x* et *y* ont été déclarées à l'aide du mot clé *private*. Ceci signifie qu'elles sont accessibles uniquement par les méthodes de classe *Point*. Il en est de même pour la méthode *check()*. La donnée membre *l* est par contre accessible de l'extérieur, comme le montre l'appel *pt.l*.

Remarque : Il existe en fait quatre déclarations de droit d'accès : *private*, pas de déclaration, *protected*, et *public*. Les différences entre ces déclarations seront abordées dans les TP suivants.

Constructeurs

Au cours de l'exemple précédent, vous avez eu recours à l'opérateur *new* et le constructeur *Point()* pour allouer l'objet *pt* instance de la classe *Point*. Vous avez utilisé ici le constructeur par défaut. Rajoutez ce bloc de code à la définition de votre classe *Point* :

```
Point() {}
```

Vous venez ici de redéfinir le constructeur par défaut de la classe *Point*. En l'absence de constructeur défini dans la classe *Point*, ce constructeur est en fait déclaré implicitement. Mettre en œuvre la classe *Point* de la façon suivante :

```
Point pt;
pt = new Point();
pt.print();
System.out.println(new Point());
new Point().print();
```

A travers ce court exemple vous avez vu la mise en œuvre du constructeur *Point()* via l'utilisation de l'opérateur *new*. La commande *new Point()* alloue un objet de type *Point* et retourne sa référence. Cette référence peut être ensuite stockée dans une variable de type *Point* (ici *pt*). Le stockage de cette référence peut également être ignorée, et l'objet utilisé « à usage unique » à un instant donné du programme comme le montre l'exemple de code : *new Point().print();*

Redéfinir votre classe de la façon suivante :

```
class Point
        {
        private int x,y;

        private void check()
                {if(x<0) x=0; if(y<0) y=0;}

        Point(int v)
                {x = y = v; check();}

        Point(int xi, int yi)
                {x = xi; y= yi; check();}

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

        void set(int xi,int yi)
                {x = xi; y= yi; check();}

        int getX() {return x;}
        int getY() {return y;}

        void print()
                {System.out.println(x+";"+y);}
        }
```

Mettre en oeuvre cette classe de la façon suivante:

```
Point pt;
pt = new Point(4);
pt.print();
pt = new Point(4,5);
pt.print();
```

Vous avez ici redéfini deux constructeurs de la classe *Point*. Que concluez-vous sur l'utilisation du constructeur par défaut dans ce cas ? pour la suite du TP, re-implémenter le constructeur *Point()* avec les constructeur *Point(int)*, et *Point(int,int)*.

Affectation, passage par argument, et comparaison d'objets

Comme nous l'avons expliqué lors de notre partie sur les constructeurs, la manipulation des objets en Java se fait par l'intermédiaire de références. Implémenter et tester le code suivant basé sur l'utilisation d'objets de type *Point*. Que concluez-vous sur l'affectation et le passage par référence d'objets.

```
void ex(Point pt) {
        pt.set(1,1);
        }

void test() {
        Point pt1, pt2;
        pt1 = new Point();
        pt2 = pt1;
        pt1.set(2,3);
        pt2.print();

        ex(pt2);
        pt1.print();
        }
```

Faîtes de même pour le code suivant mettant en œuvre la comparaison d'objets :

```
Point pt1 = new Point();
Point pt2 = new Point();
System.out.println(pt1 == pt2);
System.out.println(pt1 != pt2);
System.out.println(pt1 == pt1);
```

3) Premières notions d'héritage

Mise en œuvre de l'héritage

Basé sur la classe *Point* précédente, définir la classe *LPoint* (labelled Point) de la façon suivante:

```
class LPoint extends Point    {
        private char l;

        LPoint() {}

        LPoint(int v, char li)
                {super(v); l = li;}

        LPoint(int xi, int yi, char li)
```

```
        {super(xi,yi); l = li;}

    void setL(int xi,int yi,char li)
        {set(xi,yi); l=li;}

    char getL() {return l;}

    void printL()
        {System.out.println(getX()+";"+getY()+":"+l);}

    }
```

Mettre en oeuvre cette classe de la façon suivante :

```
LPoint pt = new LPoint();
pt.printL();
pt.setL(5,2,'b'); pt.printL();
pt.move(-6,4); pt.printL();
```

Vous avez défini ici votre première relation d'héritage entre la classe *LPoint* et *Point* à l'aide du mot clé *extends*. A travers cette nouvelle classe, vous avez encapsulé et formalisé l'utilisation de la donnée membre *l* (le label du point).

Dans la classe *LPoint* vous n'avez plus accès aux données membres *x* et *y* de l'objet *Point*. En effet, une classe dérivée n'a pas accès aux données privées de sa classe mère. Vous n'avez également plus accès à la méthode *check()* privée de la classe mère. Vous avez accès cependant à l'ensemble des méthodes non déclarées *private*, comme la méthode *move()* par exemple.

En Java, un objet dérivé doit impérativement prendre en charge la construction de l'objet père. Cette prise en charge est assurée par l'appel des constructeurs de l'objet père via le mot clé *super*. Cette instruction est toujours la première exécutée dans le constructeur d'une classe dérivée. Commenter les appels des constructeurs de l'objet père via le mot clé *super*. Vous pouvez pour cela baliser vos constructeurs avec des fonctions d'affichage de la manière suivante :

```
    LPoint() { System.out.println("1");}
```

Redéfinir la méthode *printL()* de la classe *LPoint* de la façon suivante, commenter.

```
super.print();
System.out.println(":"+l);
```

La classe racine *Object*

Mettre en oeuvre la classe *Point* de la façon suivante:

```
Point pt = new Point();
System.out.println(pt.toString());
```

Vous avez fait appel ici à une méthode *toString()* que vous n'avez pas définie dans votre classe *Point*. Cette méthode est en fait une méthode de la classe racine *Object*. En effet, toute classe en Java hérite implicitement de cette classe. Vous avez donc accès à toutes les méthodes de la classe *Object* quelque soit l'objet que vous créez.
Aller consulter dans l'API specification la documentation de la classe *Object* afin de commenter l'action de cette ligne de code.

```
System.out.println(pt.getClass().getName());
```

Garbage collector

La notion de destructeur n'existe pas en Java. La démarche employée en Java est un mécanisme de gestion automatique de la mémoire connu sous le nom de garbage collector (ou ramasse-miettes).
Le principe d'utilisation du garbage collector est basé sur l'analyse des liens entre références et objets alloués. Dès qu'un objet n'est plus référencé (sortie d'une boucle contenant une référence locale par exemple) il devient alors candidat au garbage collector. Cette candidature se traduit par l'appel de la méthode *finalize()* héritée de la classe racine *Object*. Le garbage collector est automatiquement déclenché par la machine virtuelle Java selon l'état de la mémoire du système, il désalloue alors tous les objets candidats.

Il est cependant possible de forcer le déclenchement du garbage collector. Implémenter, la classe suivante:

```
class MyOb {
        MyOb() {print();}

        String getID()
                {return Integer.toHexString(hashCode());}
        void print()
                {System.out.println(getID());}
        protected  void finalize()
                {System.out.println("finalize: "+getID());}
        }
```

Mettre en oeuvre cette classe de la façon suivante:

```
MyOb my1 = new MyOb();
new MyOb();
MyOb my2 = new MyOb(); my2=null;
System.gc();
```

System.gc() force le déclenchement du garbage collector, on admettra ici cette écriture (elle est abordée dans la suite de ce TP). Aller vérifier la déclaration de la méthode *finalize()* dans la classe *Object*, commenter l'exécution de ce programme.

4) Règles d'écriture des données membres et des méthodes

Déclaration et initialisation des données membres

Comme dans de « nombreux » langages de programmation objet, une déclaration d'une donnée membre (type primitif ou objet) d'une classe en Java se présente sous la forme suivante :

```
Class NomClasse {
        type variable ;
        }
```

Une donnée membre se caractérise donc par la déclaration de son type (type primitif ou objet), et la déclaration d'un nom de variable l'identifiant. Implémenter la classe suivante et la mettre en œuvre :

```
class Point0 {
       private int x,y;
       private char l;
       void print()
              {System.out.println(x+";"+y+":"+l);}
       }
```

Vous pouvez constater que les données membres *x* et *y* de l'objet type *Point0* ont été initialisées par défaut à 0, et la donnée membre *l* à ∅. En effet, la création d'un objet entraîne systématiquement l'initialisation de ses données membres, comme le montre le tableau suivant :

Types de données	Valeur par défaut
boolean	False
char	∅
byte, short, int, long	0
float, double	0.0
Objet	une référence

Dans ce tableau, on voit que les données de types primitifs sont initialisés. Les données de types objets elles sont initialisées par des références, mais les objets sont non alloués. C'est donc au programmeur Java de prendre en charge dans son programme les initialisations non nulles des types primitifs, et l'allocation des objets. Implémenter, et tester les classes suivantes :

```
class Point1 {
       private int x=1,y=1;
       private Object o = new Object();
       void print()
              {System.out.println(x+";"+y+":"+o);}
       }
class Point2 {
       private int x,y;
       private Object o;

       Point2()
              {x = y = 1; o = new Object();}

       void print()
              {System.out.println(x+";"+y+":"+o);}
       }
```

Comme vous pouvez le voir, il existe deux possibilités pour un programmeur Java de prise en charge des initialisations des types primitifs, et des allocations d'objets. Par la suite, habituer vous à utiliser la deuxième *Point2*. Elle constitue en effet un mode de programmation implicite en P.O.O. Il est du rôle naturel du constructeur d'assurer l'initialisation et l'allocation des données membres.

Il existe enfin en Java une possibilité pour l'initialisation définitive des données membres à l'aide du mot clé *final*. Re-implémenter la classe *Point0* de la manière suivante, mettre en œuvre la méthode *print()*, tenter d'affecter la donnée membre *l*, et commenter.

```
class Point0 {
       private int x,y;
       private final char l = 'a';

       void print()
              {System.out.println(x+";"+y+":"+l);}
       }
```

Déclaration des méthodes

Comme dans de « nombreux » langage de programmation, une déclaration de méthode de classe en Java se présente sous la forme suivante :

type nomMéthode(arguments effectifs)

Implémenter les méthodes suivantes dans la classe *Point*:

```
double distance(int xi, int yi)
        {return (x-xi)*(x-xi) + (y-yi)*(y-yi);}
```

Les arguments *xi yi* de type *int* figurant dans l'en-tête des la méthode *distance()* sont qualifiés d'arguments effectifs de la méthode. En Java, comme dans de « nombreux » langage de programmation, c'est le couple {*nomMéthode, arguments effectifs}* qui constitue la « signature » de la méthode, implémenter de nouvelles méthodes *distance()* de la façon suivante :

```
int distance(int xi, int yi)
        {return (x-xi)*(x-xi) + (y-yi)*(y-yi);}

double distance(int v)
        {return (x-v)*(x-v) + (y-v)*(y-v);}
```

Vous avez fait ici deux surcharges de la méthode *distance()*. La première surcharge déclenche une erreur à la compilation de votre code. En effet cette surcharge est erronée car le couple {*nomMéthode, arguments effectifs}* n'a pas été modifié par rapport à la première définition de la méthode *distance()*. Par opposition, la deuxième surcharge modifie le couple {*nomMéthode, arguments effectifs}*, et constitue donc une surcharge correctement formatée.

En Java, comme dans de « nombreux » langage de programmation, le passage de variables en argument d'une méthode se fait par recopie de ces variables :

- En ce qui concerne les données de types primitifs, elles sont recopiées vers les arguments effectifs de la méthode (on parle de passage par valeurs).
- En ce qui concerne les données de types objets, comme nous l'avons présenté précédemment, ceux-ci se manipulent par références. Ce sont donc les références qui sont recopiées vers les arguments effectifs de la méthode (on parle de passage par références)

Mettre en œuvre cette méthode de la façon suivante, que concluez-vous sur la récupération des arguments :

```
Point pt = new Point(1);
double d = pt.distance(5,4);
System.out.println(d);
pt.distance(5);
```

Mettre en œuvre cette méthode de la façon suivante, commenter les conversions de types liées aux opérateurs de Cast implicitement mis en œuvre lors du passage des variables à la méthode.

```
Point pt = new Point(1);
byte b=0;long q=0;
System.out.println(pt.distance(b,0));
System.out.println(pt.distance(b+3,0));
System.out.println(pt.distance(0,(int)q));
```

Implémenter cette classe et la mettre en oeuvre. Comparer et commenter cet exemple avec l'exemple d'affectation et de passage par argument des objets présenté précédemment dans ce TP.

```
void ex(Point pt)
        {pt = null;}

void test() {
        Point pt = new Point();
        System.out.println(pt);
        ex(pt);
        System.out.println(pt);
        }
```

Récursivité des méthodes

Java autorise la récursivité des appels des méthodes. Cette récursivité peut prendre deux formes :

- directe : une méthode s'appelle elle-même
- croisée : une méthode initiale appelle une méthode, qui à son tour appelle la méthode initiale

On se propose d'étudier ici la récursivité directe. Implémenter la classe suivante et la mettre en œuvre. Comparer et commenter les trois méthodes de calcul factoriel, en terme de temps de calcul et d'allocation mémoire.

```
class Util {
        long fac0(long n) {
                long r=1;
                for(long i=1;i<=n;i++)
                        r *= i;
                return r;
                }
        long fac1(long n) {
                if (n>1) return fac1(n-1)*n;
                else return 1;
                }
        long fac2(long n) {
                long r;
                if (n<=1) r = 1;
                else r = fac2(n-1)*n;
                return r;
```

```
        }
    }
```

Auto référence à l'aide du mot clé *this*

Le mot clé *this* en Java permet de faire référence à l'objet dans sa globalité. *this* correspond donc à la référence de l'objet dans lequel lequel il est utilisé. La classe suivante donne un exemple d'utilisation du mot clé *this*. Implémenter et mettre en œuvre la classe suivante, commenter :

```
class MyClass1 {
        int a=-1;

        void init()
                {a=1;}
        void clear()
                {a=-1;}

        void print() {
                init();
                System.out.println(a);
                clear();

                this.init();
                System.out.println(this.a);
                this.clear();

                System.out.println(this);
                System.out.println(toString());
                }
        }
```

this peut être utilisé à deux niveaux :

- Pour l'appel des méthodes et l'accès aux données membres, *this* est alors utilisé en cas d'ambiguïté lors de la désignation des données membres.
- Pour l'appel des constructeurs.

La classe suivante donne un exemple « pratique » d'utilisation du mot clé *this*. Implémenter et mettre en œuvre cette classe, commenter :

```
class Point4
        {
        private int x,y;

        Point4()
                {x = y = 1;}

        Point4(int v) {
                this();
                if(v!=1) x = y = v;
                }

        Point4(int x, int y) {
                this(x);


                if(x != y) {
                        this.x = x;
                        this.y= y;
                        }
                }

        void print()
                {System.out.println(x+";"+y);}
        }
```

Déclaration *static* des données membres et des méthodes

Il est possible en Java de définir des données membres et des méthodes de classe existantes en un seul exemplaire quelque soit le nombre d'objets instances d'une même classe. De même, ces données membres et ces méthodes peuvent être appelées indépendamment de toute allocation.

Implémenter la classe suivante :

```
class MyClass2 {
        static int n;
        void print()
                {System.out.println(n);}
        }
```

La mettre en œuvre de la façon suivante, commenter :

```
MyClass2 my1 = new MyClass2();
MyClass2 my2 = new MyClass2();

my1.print(); my2.print();
my2.n = 3;
my1.print(); my2.print();
System.out.println(my1 + " " + my2);

MyClass2.n = 10;
MyClass2.print();
```

```
my2.print();
```

Re-définir la classe de la façon suivante :

```
class MyClass2
        {
        static int n;

        static void print()
                {System.out.println(n);}
        }
```

Mettre alors cette classe en œuvre de la façon suivante, commenter.

```
MyClass2 my = new MyClass2();
my.print();
MyClass2.n = 10;
MyClass2.print();
my.print();
```

A travers cet exemple, vous pouvez constater qu'une donnée membre ou une méthode déclarée statique peut être utilisée sans allocation d'objet préalable. Cette déclaration statique peut être vue comme une déclaration globale (un peu à la manière du C et du C++). Cependant, la déclaration de données membres et/ou de méthodes statiques dans une classe restreint l'utilisation de celles-ci. Implémenter et commenter la classe suivante :

```
class MyClass3 {
        static int n1; //use it into f1 and f2
        int n2; //use it into f2 only

        static void f1() // use it into f2
                {n1++; n1 *= 2;}
        void f2() // use it alone {
                n2++; n2 *= 2;
                f1(); n2 += n1;
                }
        }
```

Au cours de vos différents programmes Java vous avez utilisé à plusieurs reprises des données membres et/ou des méthodes déclarées statiques, comme la méthode *main()* par exemple. Maintenant que vous avez appréhendé la notion de déclaration statique, aidez vous de l'API specification afin d'expliquer les déclarations suivantes :

1. `System.out.println()`
2. `Integer.toHexString()`
3. `System.gc()`

5) Communication entre classes

Introduction

Dans une application Java, il est souvent nécessaire de faire communiquer des objets instances de classes différentes entre eux. Il existe deux moyens pour un programmeur Java de réaliser cette communication :

- communication par classes internes
- communication par échange de références

Classes internes

Java permet de définir des classes internes, c'est-à-dire définir une classe à l'intérieur d'une définition d'une autre classe. Implémenter et commenter les classes suivantes :

```
class MyClass4 {
        class I1 {}
        I1 i1 = new I1(), i2;

        void f1()
                {i2 = new I1();I1 i3 = new I1();}

        void f2() {
                class I2{}
                I2 i = new I2();
                }
        }
```

Comme vous pouvez le voir sur cet exemple, la définition de classe interne « complique » la déclaration des classes. Cette déclaration interne a cependant deux avantages réciproques :

- Un objet d'une classe interne a toujours accès aux données membres et méthodes de son objet de classe externe (englobante)
- Un objet d'une classe externe a toujours accès aux données membres et méthodes de son/ses objet(s) de classe(s) interne(s)

Implémenter les classes suivantes, faites appel aux méthodes *f1()* et *f2()*, commenter :

```
class MyClass5 {
        I i = new I();

        private int n1;
```

```
        void f1() {n1++; i.n2++; i.print2();}
        private void print1()
                {System.out.println("1:"+n1);}
        class I  {
                private int n2;
                void f2() {n2++; n1++; print1();}
                private void print2()
                        {System.out.println("2:"+n2);}
                }
        }
```

Remarque : Vous avez également la possibilité de définir vos classes internes de façon statique. Ceci permet d'utiliser une classe interne à l'extérieur de sa classe externe, mais impose les contraintes liées à la déclaration statique exposées précédemment.

Communication par échange de références

Une autre façon plus « élégante » d'effectuer une communication entre objets de classes différentes et d'échanger les références des objets. Implémenter les classes suivantes :

```
class Class1 {
        private int n; int c;

        void inc(int i)
                {n+=i;}

        int get() {return n;}
        void set(int n) {this.n = n;}
        void print()
                {System.out.println(n+":"+c);}

        Class2 my;
        Class1(Class2 my) {this.my = my;}
        void exchange()
                {int v = n; n = my.get(); my.set(v); my.c++;}
        }

class Class2 {
        private int n; int c;

        void inc(int i)
                {n+=i;}

        int get() {return n;}
        void set(int n) {this.n = n;}
        void print()
                {System.out.println(n+":"+c);}

        Class1 my;
        Class2(Class1 my) {this.my = my;}

        void exchange()
                {int v = n; n = my.get(); my.set(v); my.c++;}
        }
```

Mettre en œuvre ces classes de la façon suivante, commenter.

```
Class1 my1 = null; Class2 my2 = null;
my1 = new Class1(my2 = new Class2(my1));

my1.inc(5); my2.inc(10);
my1.print(); my2.print();

my1.exchange();
my1.print(); my2.print();

my1.exchange();
my1.print(); my2.print();
```

De quelle autre façon qu'un passage par constructeur pourriez vous échanger les références ?

6) Utilisation de package

Introduction

La notion de package en Java correspond à un regroupement de classes, sous un identificateur commun (correspondant au nom du package). Cette notion facilite le développement et la cohabitation d'application « conséquente », en isolant l'ensemble des classes existantes. Dans l'API specification, la fenêtre haut-droit donne la liste des packages standards de la plate-forme Java 2 SDK. Pour importer un package dans un fichier Java, il suffit de déclarer ce package à l'aide de l'instruction *import* en en-tête de votre fichier de la façon suivante :

```
import java.lang.*;
```

Sans le savoir, vous avez utilisé ce package *java.lang* à plusieurs reprises dans vos programmes Java. En effet, ce package standard *java.lang* est en fait implicitement déclaré dans les fichiers Java (Nous l'utilisons ici à titre d'exemple). Si vous sélectionnez ce package dans l'API specification, vous y trouverez l'ensemble des classes que vous avez utilisées au cours de vos programmes : *String*, *System*, *Integer*, etc.

De même, vous avez déjà défini des packages à plusieurs reprises dans vos programmes Java. En effet, lorsque vous compilez différentes classes stockées en (.class) dans un même répertoire, vous constituez un package local (sans nom).

Dans notre déclaration exemple, l'instruction * définit que toutes les classes du package *java.lang* sont importées. Il est en effet possible de filtrer les classes que l'on souhaite importer en les sélectionnant individuellement, comme par exemple :

```
import java.lang.String;
```

L'importation de toutes les classes d'un package n'augmente ni la taille du byte code Java, ni le temps d'exécution du programme. Le filtrage des classes lors de l'importation d'un package n'a d'utilité que de limiter les ambiguïtés en cas de classes de noms similaires entre packages différents importés.

L'instruction . est un séparateur entre les noms de package, de sous packages, et l'instruction *. En effet, dans notre exemple le package *lang* est un sous-package du package *java*. Dans l'API specification, Vous verrez qu'il existe trois packages racines : *java*, *javax*, et *org*. L'exemple suivant donne une hiérarchie de package à trois étages.

```
import java.util.zip.*;
import java.util.jar.*;
```

Première mise en oeuvre

On se propose ici de faire une première importation de package, et une première utilisation de classe de package. Implémenter et commenter le code suivant, aidez-vous de l'API specification :

```
import java.util.*;

public class PackageUse1
	{
	public static void main(String args[])
		{
		Random r = new Random();
		System.out.println(r.nextInt());
		}
	}
```

Gestion des ambiguïtés

Comme nous l'avons présenté en introduction, l'utilisation des packages permet de lever les ambiguïtés entre classes de noms similaires entre packages différents. On se propose d'illustrer cette propriété ici en confrontant une classe *Vector* locale avec la classe *Vector* du package *java.util*. Implémenter le code suivant et le mettre en oeuvre:

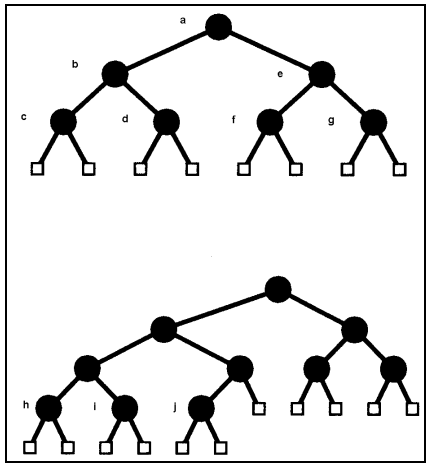
```
import java.util.*;

class Vector   {
	private Object t[];
	private int p;

	Vector()
		{t = new Object[100];}

	void add(Object o) {
		if(p<100) {t[p] = o; p++;}
		System.out.println("add");
		}

	Object elementAt(int i)
		{if (i<=p) return t[i]; else return null;}
	}

class Stack {
	private java.util.Vector v1;
	private Vector v2;
	int p;

	Stack() {
		v1 = new java.util.Vector();
		v2 = new Vector();
	}

	void push(Object o) {
		v1.add(o);
		v2.add(o);
		System.out.println("push");
		p++;
		}
	Object peek() {
		return v2.elementAt(p-1);
		}
	}

public class PackageUse2 {
	public static void main(String args[])
```

```
        {
        Stack st = new Stack();
        st.push(new Object());
        System.out.println(st.peek());
        }
}
```

Que concluez vous sur la déclaration de *v1* en *java.util.Vector,* commenter. Recommencez une première fois une compilation/exécution de ce code en mettant la classe *Stack* en commentaire. Re-faîtes une seconde fois une compilation/exécution mais en supprimant au préalable le fichier *Stack.class* dans votre répertoire local. Que concluez vous sur une déclaration éventuelle de la variable *st* en *java.util.Stack* ?

7) Algorithmes de parcours d'arbres binaires

Dans cette partie, on se propose de mettre en œuvre les notions abordées dans ce TP pour l'implémentation d'algorithmes de parcours d'arbres binaires. L'arbre est une structure de données informatique classique. Par exemple, la structuration des fichiers d'un disque de stockage se fait sous forme d'arbre. L'arbre binaire est une spécialisation de la structure de données arbre, dans lequel chaque noeud de l'arbre est connecté à 0-2 nœuds fils. La figure ci-après donne deux exemples d'arbres binaires composés respectivement de 7 et 10 nœuds. L'arbre à 10 nœuds à pour racine l'arbre à 7 nœuds, plus 3 nœuds supplémentaires : h, i, et j.



On se propose dans une première étape d'implémenter la structure de données arbre binaire. Celle-ci se limite à la conception d'une classe *Node* constituée de deux références internes pour les nœuds fils gauche (*l*) et droit (*r*). Implémenter la classe suivante en la complétant, la mettre en œuvre.

```
class Node {
        private Node l,r;
        Node(){…}
        Node getL(){…}
        Node getR(){…}
        void set(Node l, Node r) {…}
        }
```

On se propose de gérer automatiquement à la création de chaque nœud un label incrémental de type caractère. Reprenez votre classe *Node*, déclarez un label statique et un label dynamique. Votre constructeur devra affecter votre label dynamique avec le label statique, et incrémenter le label statique. Ainsi, l'état du label statique sera transmis entre tous les objets instances de la classe *Node*. Définissez dans votre classe *Node* une méthode *getID()* permettant d'afficher le label dynamique. Prévoir également une méthode statique *clearID()* pour la remise à « zéro » du label statique.

Définir une classe *Search* de la façon suivante. Dans une première étape, initialiser le constructeur de façon à construire les arbres binaires de 7 et 10 nœuds présentés sur la figure précédente avec pour nœud racine respectivement *r1* et *r2*. Entre les constructions des deux arbres, utiliser la méthode *clearID()* pour la remise à zéro du label statique.

```
class Search   {
        Node r1, r2;
        Search() {…}
        void traverse1(Node n) {…}
        void traverse2(Node n) {…}
        void traverse3(Node n) {…}
        }
```

Pour les arbres binaires, il existe 2 liens et donc 3 parcours élémentaires pour passer par les nœuds :

- parcours préfixe : nœud père, nœud fils gauche, nœud fils droit
- parcours infixe : nœud fils gauche, nœud père, nœud fils droit
- parcours postfixe : nœud fils gauche, nœud fils droit, nœud père

On se propose d'implémenter 3 fonctions récursives pour la mise en œuvre de ces parcours : *traverse1()*, *traverse2()*, et *traverse3()*. Dans ces fonctions, le passage par un nœud se traduit par l'appel de sa fonction *getID()* pour l'affichage de son label. Implémentez ces trois fonctions, pour vérifier vos fonctions, les listes suivantes donnent les résultats pour les 3 types de parcours concernant l'arbre à 7 nœuds de la figure précédente. Indiquer les résultats concernant l'arbre à 10 nœuds.

- préfixe : a, b, c, d, e, f, g
- infixe : c, b, d, a, f, e, g
- postfixe : c, d, b, f, g, e, a