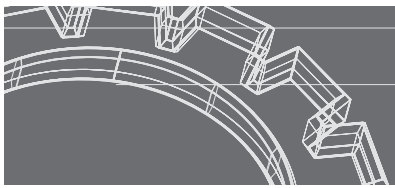
Chapitre 3

Les classes et les objets



Connaissances requises

- Notion de classe : définition des champs et des méthodes, accès privés ou publics aux membres, utilisation d'une classe
- Mise en oeuvre d'un programme comportant plusieurs classes, à raison d'une ou plusieurs classes par fichier source
- Notion de constructeur ; règles d'écriture et d'utilisation
- Les différentes étapes de la création d'un objet : initialisation par défaut, initialisation explicite, appel du constructeur ; cas particulier des champs déclarés avec l'attribut *final*
- Affectation et comparaison d'objets
- Notion de ramasse-miettes
- Règles d'écriture d'une méthode ; méthode fonction, arguments muets ou effectifs, règles de conversion des arguments effectifs, propriétés des variables locales
- Champs et méthodes de classe ; initialisation des champs de classe, bloc d'initialisation statique
- Surdéfinition de méthodes
- Le mot clé *this* ; cas particulier de l'appel d'un constructeur au sein dun autre constructeur
- Récursivité des méthodes
- Mode de transmission des arguments et de la valeur de retour
- Objets membres
- Paquetages

23 Création et utilisation d'une classe simple

Réaliser une classe *Point* permettant de représenter un point sur un axe. Chaque point sera caractérisé par un nom (de type *char*) et une abscisse (de type *double*). On prévoira :

- un constructeur recevant en arguments le nom et l'abscisse d'un point,
- une méthode *affiche* imprimant (en fenêtre console) le nom du point et son abscisse,
- une méthode *translate* effectuant une translation définie par la valeur de son argument.

Écrire un petit programme utilisant cette classe pour créer un point, en afficher les caractéristiques, le déplacer et en afficher à nouveau les caractéristiques.

Solution

Ici, notre programme d'essai (méthode *main*) est séparé de la classe *Point*, mais placé dans le même fichier source. La classe *Point* ne peut donc pas être déclarée publique. Rappelons que, dans ces conditions, elle reste utilisable depuis n'importe quelle classe du paquetage par défaut.

```
class Point
{ public Point (char c, double x)   // constructeur
  { nom = c ;
    abs = x ;
  }
  public void affiche ()
  { System.out.println ("Point de nom " + nom + "  d'abscisse " + abs) ;
  }
  public void translate (double dx)
  { abs += dx ;
  }
  private char nom ;     // nom du point
  private double abs ;   // abscisse du point
}
public class TstPtAxe
{ public static void main (String args[])
  { Point a = new Point ('C', 2.5) ;
    a.affiche() ;
    Point b = new Point ('D', 5.25) ;
    b.affiche() ;
    b.translate(2.25) ;
    b.affiche() ;
  }
}
```

```
Point de nom C  d'abscisse 2.5
Point de nom D  d'abscisse 5.25
Point de nom D  d'abscisse 7.5
```

24 Initialisation d'un objet

Que fournit le programme suivant ?

```
class A
{ public A (int coeff)
  { nbre *= coeff ;
    nbre += decal ;
  }

  public void affiche ()
  { System.out.println ("nbre = " + nbre + "  decal = " + decal) ;
  }
  private int nbre = 20 ;
  private int decal ;
}

public class InitChmp
{ public static void main (String args[])
  { A a = new A (5) ; a.affiche() ;
  }
}
```

Solution

La création d'un objet de type *A* entraîne successivement :

- l'initialisation par défaut de ses champs *nbre* et *decal* à une valeur "nulle" (ici l'entier 0),
- l'initialisation explicite de ses champs lorsqu'elle existe ; ici *nbre* prend la valeur 20,
- l'appel du constructeur : *nbre* est multiplié par la valeur de *coeff* (ici 5), puis incrémenté de la valeur de *decal* (0).

En définitive, le programme affiche :

```
nbre = 100  decal = 0
```

25 Champs constants

Quelle erreur a été commise dans cette définition de classe ?

```
class ChCt
{ public ChCt (float r)
  { x = r ;
  }
      .....
  private final float x ;
  private final int n = 10 ;
  private final int p ;
}
```

Solution

Le champ *p* déclaré *final* doit être initialisé au plus tard par le constructeur, ce qui n'est pas le cas. En revanche, les autres champs déclarés *final* sont correctement initialisés, *n* de façon explicite et *x* par le constructeur.

26 Affectation et comparaison d'objets

Que fournit le programme suivant ?

```
class Entier
{ public Entier (int nn)    { n = nn ; }
  public void incr (int dn) { n += dn ; }
  public void imprime ()    { System.out.println (n) ; }
  private int n ;
}
public class TstEnt
{ public static void main (String args[])
  { Entier n1 = new Entier (2) ; System.out.print ("n1 = ") ; n1.imprime() ;
    Entier n2 = new Entier (5) ; System.out.print ("n1 = ") ; n2.imprime() ;
    n1.incr(3) ;               System.out.print ("n1 = ") ; n1.imprime() ;
    System.out.println ("n1 == n2 est " + (n1 == n2)) ;
    n1 = n2 ;  n2.incr(12) ;    System.out.print ("n2 = ") ; n2.imprime() ;
                               System.out.print ("n1 = ") ; n1.imprime() ;
    System.out.println ("n1 == n2 est " + (n1 == n2)) ;
  }
}
```

Solution

```
n1 = 2
n1 = 5
n1 = 5
n1 == n2 est false
n2 = 17
n1 = 17
n1 == n2 est true
```

L'opérateur == appliqué à des objets compare leurs références (et non leurs valeurs). C'est pourquoi la première comparaison (*n1 == n2)* est fausse alors que les objets ont la même valeur. La même reflexion s'applique à l'opérateur d'affectation. Après exécution de *n1 = n2*, les références contenues dans les variables *n1* et *n2* sont les mêmes. L'objet anciennement référencé par *n2* n'étant plus référencé par ailleurs, il devient candidat au ramasse-miettes.

Dorénavant *n1* et *n2* référencent un seul et même objet. L'incrémentation de sa valeur par le biais de *n1* se retrouve indifféremment dans *n1.imprime* et dans *n2.imprime*. De même, la comparaion *n1 == n2* a maintenant la valeur vrai.

27 Méthodes d'accès aux champs privés

Soit le programme suivant comportant la définition d'une classe nommée *Point* et son utilisation :

```
class Point
{ public Point (int abs, int ord)      { x = abs ;  y = ord ; }
  public void deplace (int dx, int dy) { x += dx ;  y += dy ; }
  public void affiche ()
  { System.out.println ("Je suis un point de coordonnees " + x + " " + y) ;
  }
  private double x ;   // abscisse
  private double y ;   // ordonnee
}
public class TstPnt
{ public static void main (String args[])
  { Point a ;
    a = new Point(3, 5) ;        a.affiche() ;
    a.deplace(2, 0) ;            a.affiche() ;
    Point b = new Point(6, 8) ;  b.affiche() ;
  }
}
```

Modifier la définition de la classe *Point* en supprimant la méthode *affiche* et en introduisant deux méthodes d'accès nommées *abscisse* et *ordonnee* fournissant respectivement l'abscisse et l'ordonnée d'un point. Adapter la méthode *main* en conséquence.

Solution

```
class Point
{ public Point (int abs, int ord)      { x = abs ;  y = ord ; }
  public void deplace (int dx, int dy) { x += dx ;  y += dy ; }
  public double abscisse () { return x ; }
  public double ordonnee () { return y ; }
  private double x ;   // abscisse
  private double y ;   // ordonnee
}
public class TstPnt1
{ public static void main (String args[])
  { Point a ;
    a = new Point(3, 5) ;
    System.out.println ("Je suis un point de coordonnees "
                        + a.abscisse() + " " + a.ordonnee()) ;
    a.deplace(2, 0) ;
    System.out.println ("Je suis un point de coordonnees "
                        + a.abscisse() + " " + a.ordonnee()) ;
    Point b = new Point(6, 8) ;
    System.out.println ("Je suis un point de coordonnees "
                        + b.abscisse() + " " + b.ordonnee()) ;
  }
}
```

Remarque

Cet exemple était surtout destiné à montrer que les méthodes d'accès permettent de respecter l'encapsulation des données. Dans la pratique, la classe disposera probablement d'une méthode *affiche* en plus des méthodes d'accès.

28 Conversions d'arguments

On suppose qu'on dispose de la classe *A* ainsi définie :

```
class A
{ void f (int n, float x) { ..... }
  void g (byte b) { ..... }
    .....
}
```

Soit ces déclarations :

```
A a ; int n ; byte b ; float x ; double y ;
```

Dire si les appels suivants sont corrects et sinon pourquoi.

```
a.f (n, x) ;
a.f (b+3, x) ;
a.f (b, x) ;
a.f (n, y) ;
```

```
a.f (n, (float)y) ;
a.f (n, 2*x) ;
a.f (n+5, x+0.5) ;
a.g (b) ;
a.g (b+1) ;
a.g (b++) ;
a.g (3) ;
```

Solution

```
a.f (n, x) ;          // OK : appel normal
a.f (b+3, x) ;        // OK : b+3 est déjà de type int
a.f (b, x) ;          // OK : b de type byte sera converti en int
a.f (n, y) ;          // erreur : y de type double ne peut être converti en float
a.f (n, (float)y) ;   // OK
a.f (n, 2*x) ;        // OK : 2*x est de type float
a.f (n+5, x+0.5) ;    // erreur : 0.5 est de type double, donc x+0.5 est de
                      //  type double, lequel ne peut pas être converti en float
a.g (b) ;             // OK : appel normal
a.g (b+1) ;           // erreur : b1+1 de type int ne peut être converti en byte
a.g (b++) ;           // OK : b1++ est de type int
                      //  (mais peu conseillé : on a modifié la valeur de b1)
a.g (3) ;             // erreur : 3 de type int ne peut être convertie en byte
```

29 Champs et méthodes de classe (1)

Quelles erreurs ont été commises dans la définition de classe suivante et dans son utilisation ?

```
class A
{ static int f (int n)
  { q = n ;
  }
  void g (int n)
  { q = n ;
    p = n ;
  }
  static private final int p = 20 ;
  private int q ;
}
public class EssaiA
{ public static void main (String args[])
  { A a = new A() ; int n = 5 ;
    a.g(n) ;
```

```
        a.f(n) ;
        f(n) ;
      }
    }
```

Solution

La méthode statique *f* de *A* ne peut pas agir sur un champ non statique ; l'affectation *q=n* est incorrecte.

Dans la méthode *g* de *A*, l'affectation *q=n* n'est pas usuelle mais elle est correcte. En revanche, l'affectation *p=n* ne l'est pas puisque *p* est *final* (il doit donc être initialisé au plus tard par le constructeur et il ne peut plus être modifié par la suite).

Dans la méthode *main*, l'appel *a.f(n)* se réfère à un objet, ce qui est inutile mais toléré. Il serait cependant préférable de l'écrire *A.f(n)*. Quant à l'appel *f(n)* il est incorrect puisqu'il n'existe pas de méthode *f* dans la classe *EssaiA*[1]. Il est probable que l'on a voulu écrire *A.f(n)*.

30 Champs et méthodes de classe (2)

Créer une classe permettant de manipuler un point d'un axe, repéré par une abscisse (de type *int*). On devra pouvoir effectuer des changements d'origine, en conservant en permanence l'abscisse d'une origine courante (initialement 0). On prévoira simplement les méthodes suivantes :

- constructeur, recevant en argument l'abscisse "absolue" du point (c'est-à-dire repérée par rapport au point d'origine 0 et non par rapport à l'origine courante),
- *affiche* qui imprime à la fois l'abscisse de l'origine courante et l'abscisse du point par rapport à cette origine,
- *setOrigine* qui permet de définir une nouvelle abscisse pour l'origine (exprimée de façon absolue et non par rapport à l'origine courante),
- *getOrigine* qui permet de connaître l'abscisse de l'origine courante.

Ecrire un petit programme de test fournissant les résultats suivants :

```
Point a - abscisse = 3
   relative a une origine d'abscisse 0
Point b - abscisse = 12
   relative a une origine d'abscisse 0
On place l'origine en 3
```

1. Si la méthode *main* avait été introduite directement dans *A*, l'appel serait accepté !

```
Point a - abscisse = 0
   relative a une origine d'abscisse 3
Point b - abscisse = 9
   relative a une origine d'abscisse 3
```

Solution

L'abscisse de l'origine courante est une information qui concerne tous les points de la classe. On en fera donc un champ de classe en le déclarant *static*. De la même manière, les méthodes *setOrigine* et *getOrigine* concernent non pas un point donné, mais la classe. On en fera des méthodes de classe en les déclarant *static*.

```
class Point
{ public Point (int xx) { x = xx ; }
  public void affiche ()
  { System.out.println ("abscisse = " + (x-origine)) ;
    System.out.println ("   relative a une origine d'abscisse " + origine) ;
  }
  public static void setOrigine (int org) { origine = org ;  }
  public static int getOrigine()          { return origine ; }
  private static int origine ;   // abscisse absolue de l'origine courante
  private int x ;                // abscisse absolue du point
}

public class TstOrig
{ public static void main (String args[])
  { Point a = new Point (3) ;  System.out.print ("Point a - ") ; a.affiche() ;
    Point b = new Point (12) ; System.out.print ("Point b - ") ; b.affiche() ;
    Point.setOrigine(3) ;
    System.out.println ("On place l'origine en " + Point.getOrigine()) ;
    System.out.print ("Point a - ") ; a.affiche() ;
    System.out.print ("Point b - ") ; b.affiche() ;
  }
}
```

31 Champs et méthodes de classe (3)

Réaliser une classe qui permet d'attribuer un numéro unique à chaque nouvel objet créé (1 au premier, 2 au suivant...). On ne cherchera pas à réutiliser les numéros d'objets éventuellement détruits. On dotera la classe uniquement d'un constructeur, d'une méthode *getIdent* fournissant le numéro attribué à l'objet et d'une méthode *getIdentMax* fournissant le numéro du dernier objet créé.

Écrire un petit programme d'essai.

Solution

Chaque objet devra disposer d'un champ (de préférence privé) destiné à conserver son numéro. Par ailleurs, le constructeur d'un objet doit être en mesure de connaître le dernier numéro attribué. La démarche la plus naturelle consiste à le placer dans un champ de classe (nommé ici *numCour*). La méthode *getIdentMax* est indépendante d'un quelconque objet ; il est préférable d'en faire une méthode de classe.

```
class Ident
{ public Ident ()
  { numCour++ ;
    num = numCour ;
  }
  public int getIdent()
  { return num ;
  }
  public static int getIdentMax()
  { return numCour ;
  }
  private static int numCour=0 ;  // dernier numero attribué
  private int num ;               // numero de l'objet
}
public class TstIdent
{ public static void main (String args[])
  { Ident a = new Ident(), b = new Ident() ;
    System.out.println ("numero de a : " + a.getIdent()) ;
    System.out.println ("numero de b : " + b.getIdent()) ;
    System.out.println ("dernier numero " + Ident.getIdentMax()) ;
    Ident c = new Ident() ;
    System.out.println ("dernier numero " + Ident.getIdentMax()) ;
  }
}
```

Ce programme fournit les résultats suivants :

```
numero de a : 1
numero de b : 2
dernier numero 2
dernier numero 3
```

Remarque

Si l'on souhaitait récupérer les identifications d'objets détruits, on pourrait exploiter le fait que Java appelle la méthode *finalize* d'un objet avant de le soumettre au ramasse-miettes. Il faudrait alors redéfinir cette méthode en conservant les numéros ainsi récupérés et en les réutilisant dans une construction ultérieure d'objet, ce qui compliquerait quelque peu la définition de la classe. De plus, il ne faudrait pas perdre de vue qu'un objet n'est soumis au ramasse-miettes qu'en cas de besoin de mémoire et non pas nécessairement dès qu'il n'est plus référencé.

32 Bloc d'initialisation statique

Adapter la classe précédente, de manière que le numéro initial des objets soit lu au clavier[a]. On devra s'assurer que la réponse de l'utilisateur est strictement positive.

a. On pourra utiliser la méthode *lireInt* de la classe *Clavier* fournie sur le site Web d'accompagnement et dont la liste figure en Annexe D

Solution

S'il n'était pas nécessaire d'effectuer un test sur la valeur fournie au clavier, on pourrait se contenter de modifier ainsi la classe *Ident* précédente :

```
public Ident ()
{  num = numCour ;
   numCour++ ;
}
   .....
private static int numCour=Clavier.lireInt() ;  // dernier numero attribué
```

Notez cependant que l'utilisateur ne serait pas informé que le programme attend qu'il frappe au clavier.

Mais ici, l'initialisation de *numCour* n'est plus réduite à une simple expression. Elle fait donc obligatoirement intervenir plusieurs instructions et il est nécessaire de recourir à un bloc d'initialisation statique en procédant ainsi :

```
class Ident
{ public Ident ()
  { num = numCour ;
    numCour++ ;
  }
  public int getIdent()
  { return num ;
  }
  public static int getIdentMax()
  { return numCour-1 ;
  }
  private static int numCour ;   // prochain numero a attribuer
  private int num ;              // numero de l'objet
  static
  { System.out.print ("donnez le premier identificateur : ") ;
    do numCour = Clavier.lireInt() ; while (numCour <= 0) ;
  }
}
```

À titre indicatif, avec le même programme (*main*) que dans l'exercice précédent, on obtient ces résultats :

```
donnez le premier identificateur : 12
numero de a : 12
numero de b : 13
dernier numero 13
dernier numero 14
```

Remarque

1. Les instructions d'un bloc d'initialisation statique ne concernent aucun objet en particulier ; elles ne peuvent donc accéder qu'à des champs statiques. En outre, et contrairement à ce qui se produit pour les instructions des méthodes, ces champs doivent avoir été déclarés avant d'être utilisés. Ici, il est donc nécessaire que la déclaration du champ statique *numCour* figure avant le bloc statique (en pratique, on a tendance à placer ces blocs en fin de définition de classe).
2. Les instructions d'un bloc d'initialisation sont exécutées avant toute création d'un objet de la classe. Même si notre programme ne créait aucun objet, il demanderait à l'utilisateur de lui founir un numéro.

33 Surdéfinition de méthodes

Quelles erreurs figurent dans la définition de classe suivante ?

```
class Surdef
{ public void f (int n)          { ..... }
  public int f (int p)           { ..... }
  public void g (float x)        { ..... }
  public void g (final double y) { ..... }
  public void h (long n)         { ..... }
  public int h (final long p)    { ..... }
}
```

Solution

Les deux méthodes *f* ont des arguments de même type (la valeur de retour n'intervenant pas dans la surdéfinition des fonctions). Il y a donc une ambiguïté qui sera détectée dès la compilation de la classe, indépendamment d'une quelconque utilisation.

La surdéfinition des méthodes *g* ne présente pas d'anomalie, leurs arguments étant de types différents.

Enfin, les deux méthodes *h* ont des arguments de même type (*long*), le qualificatif *final* n'intervenant pas ici. La compilation signalera également une ambiguïté à ce niveau.

34 Recherche d'une méthode surdéfinie (1)

Soit la définition de classe suivante :

```
class A
{ public void f (int n)           { ..... }
  public void f (int n, int q)    { ..... }
  public void f (int n, double y) { ..... }
}
```

Avec ces déclarations :

```
A a ; byte b ; short p ; int n ; long q ; float x ; double y ;
```

Quelles sont les instructions correctes et, dans ce cas, quelles sont les méthodes appelées et les éventuelles conversions mises en jeu ?

```
a.f(n);
a.f(n, q) ;
a.f(q) ;
a.f(p, n) ;
a.f(b, x) ;
a.f(q, x) ;
```

Solution

```
a.f(n);        // appel f(int)
a.f(n, q) ;    // appel f(int, double) après conversion de q en double
a.f(q) ;       // erreur : aucune méthode acceptable
a.f(p, n) ;    // appel f(int, int) après conversion de p en int
a.f(b, x) ;    // appel f(int, double) après conversion de b en int
               //   et de x en double
a.f(q, x) ;    // erreur : aucune méthode acceptable
```

35 Recherche d'une méthode surdéfinie (2)

Soit la définition de classe suivante :

```
class A
{ public void f (byte b)   { ..... }
  public void f (int n)    { ..... }
  public void f (float x)  { ..... }
  public void f (double y) { ..... }
}
```

Avec ces déclarations :

```
A a ; byte b ; short p ; int n ; long q ; float x ; double y ;
```

Quelles sont les méthodes appelées et les éventuelles conversions mises en jeu dans chacune des instructions suivantes ?

```
a.f(b) ;
a.f(p) ;
a.f(q) ;
a.f(x) ;
a.f(y) ;
a.f(2.*x) ;
a.f(b+1) ;
a.f(b++) ;
```

Solution

```
a.f(b) ;      // appel de f(byte)
a.f(p) ;      // appel de f(int)
a.f(q) ;      // appel de f(float) après conversion de q en float
a.f(x) ;      // appel de f(float)
a.f(y) ;      // appel de f(double)
a.f(2.*x) ;   // appel de f(double) car 2. est de type double ;
              //   l'expression 2.*x est de type double
a.f(b+1) ;    // appel de f(int) car l'expression b+1 est de type int
a.f(b++) ;    // appel de f(byte) car l'expression b++ est de type byte
```

36 Recherche d'une méthode surdéfinie (3)

Soit la définition de classe suivante :

```
class A
{ public void f (int n, float x)
  { ..... }
  public void f (float x1, float x2)
  { ..... }
  public void f (float x, int n)
  { ..... }
}
```

Avec ces déclarations :

```
A a ; short p ; int n1,  n2 ; float x ;
```

Quelles sont les instructions correctes et, dans ce cas, quelles sont les méthodes appelées et les éventuelles conversions mises en jeu ?

```
a.f(n1, x) ;
a.f(x, n1) ;
a.f(p, x) ;
a.f(n1, n2) ;
```

Solution

```
a.f(n1, x) ;
```

Les méthodes *f(int, float)* et *f(float, float)* sont acceptables mais la seconde est moins bonne que la première. Il y a donc appel de *f(int, float)*.

```
a.f(x, n1) ;
```

Les méthodes *f(float, float)* et *f(float, int)* sont acceptables mais la première est moins bonne que la seconde. Il y a donc appel de *f(float, int)*.

```
a.f(p, x) ;
```

Les trois méthodes sont acceptables. La seconde et la troisième sont moins bonnes que la première. Il y a donc appel de *f(int, float)* après conversion de *p* en *int*.

```
a.f(n1, n2) ;
```

Les trois méthodes sont acceptables. Seule la seconde est moins bonne que les autres. Comme aucune des deux méthodes *f(int, float)* et *f(float, int)* n'est meilleure que les autres, il y a erreur.

37 Surdéfinition et droits d'accès

Quels résultats fournit ce programme ?

```
class A
{ public void f(int n, float x)
  { System.out.println ("f(int n, float x)      n = " + n + " x = " + x) ;
  }
  private void f(long q, double y)
  { System.out.println ("f(long q, double y)    q = " + q + " y = " + y) ;
  }
  public void f(double y1, double y2)
  { System.out.println ("f(double y1, double y2)  y1 = " + y1 + " y2 = " + y2) ;
  }
  public void g()
  { int n=1 ; long q=12 ; float x=1.5f ; double y = 2.5 ;
    System.out.println ("--- dans g ") ;
    f(n, q) ;
    f(q, n) ;
    f(n, x) ;
    f(n, y) ;
  }
}
```

```
public class SurdfAcc
{ public static void main (String args[])
  { A a = new A() ;
    a.g() ;
    System.out.println ("--- dans main") ;
    int n=1 ; long q=12 ; float x=1.5f ; double y = 2.5 ;
    a.f(n, q) ;
    a.f(q, n) ;
    a.f(n, x) ;
    a.f(n, y) ;
  }
}
```

Solution

```
--- dans g
f(int n, float x)        n = 1 x = 12.0
f(long q, double y)      q = 12 y = 1.0
f(int n, float x)        n = 1 x = 1.5
f(long q, double y)      q = 1 y = 2.5
--- dans main
f(int n, float x)        n = 1 x = 12.0
f(double y1, double y2)  y1 = 12.0 y2 = 1.0
f(int n, float x)        n = 1 x = 1.5
f(double y1, double y2)  y1 = 1.0 y2 = 2.5
```

La méthode *f(long, double)* étant privée, elle n'est accessible que depuis les méthodes de la classe. Ici, elle est donc accessible depuis *g* et elle intervient dans la recherche de la meilleure correspondance dans un appel de *f*. En revanche, elle ne l'est pas depuis *main*. Ceci explique les différences constatées dans les deux séries d'appels identiques, l'une depuis *g*, l'autre depuis *main*.

38 Emploi de this

Soit la classe *Point* ainsi définie :

```
class Point
{ public Point (int abs, int ord)      { x = abs ;  y = ord ; }
  public void affiche ()
  { System.out.println ("Coordonnees " + x + " " + y) ;
  }
  private double x ;   // abscisse
  private double y ;   // ordonnee
}
```

Lui ajouter une méthode *maxNorme* déterminant parmi deux points lequel est le plus éloigné de l'origine et le fournissant en valeur de retour. On donnera deux solutions :

- *maxNorme* est une méthode statique de *Point*,
- *maxNorme* est une méthode usuelle de *Point*.

Solution

Avec une méthode statique

La méthode *maxNorme* va devoir disposer de deux arguments de type *Point*. Ici, nous nous contentons de calculer le carré de la norme du segment joignant l'origine au point concerné. Il suffit ensuite de fournir comme valeur de retour celui des deux points pour lequel cette valeur est la plus grande. Voici la nouvelle définition de la classe *Point*, accompagnée d'un programme de test et des résultats fournis par son exécution :

```
class Point
{ public Point (int abs, int ord)      { x = abs ;  y = ord ; }
  public void affiche ()
  { System.out.println ("Coordonnees " + x + " " + y) ;
  }
  public static Point MaxNorme (Point a, Point b)
  { double na = a.x*a.x + a.y*a.y ;
    double nb = b.x*b.x + b.y*b.y ;
    if (na>nb) return a ;
          else return b ;
  }
  private double x ;   // abscisse
  private double y ;   // ordonnee
}

public class MaxNorme
{ public static void main (String args[])
  { Point p1 = new Point (2, 5) ; System.out.print ("p1 : ") ; p1.affiche() ;
    Point p2 = new Point (3, 1) ; System.out.print ("p2 : ") ; p2.affiche() ;
    Point p = Point.MaxNorme (p1, p2) ;
    System.out.print ("Max de p1 et p2 : ") ; p.affiche() ;
  }
}
```

```
p1 : Coordonnees 2.0 5.0
p2 : Coordonnees 3.0 1.0
Max de p1 et p2 : Coordonnees 2.0 5.0
```

Avec une méthode usuelle

Cette fois, la méthode ne dispose plus que d'un seul argument de type *Point*, le second point concerné étant celui ayant appelé la méthode et dont la référence se note simplement *this*.

Voici la nouvelle définition de la classe et l'adaptation du programme d'essai (qui fournit les mêmes résultats que précédemment) :

```
class Point
{ public Point (int abs, int ord)      { x = abs ;  y = ord ; }
  public void affiche ()
  { System.out.println ("Coordonnees " + x + " " + y) ;
  }
  public Point MaxNorme (Point b)
  { double na = x*x + y*y ;    // ou encore this.x*this.x + this.y*this.y
    double nb = b.x*b.x + b.y*b.y ;
    if (na>nb) return this ;
          else return b ;
  }
  private double x ;   // abscisse
  private double y ;   // ordonnee
}
public class MaxNorm2
{ public static void main (String args[])
  { Point p1 = new Point (2, 5) ; System.out.print ("p1 : ") ; p1.affiche() ;
    Point p2 = new Point (3, 1) ; System.out.print ("p2 : ") ; p2.affiche() ;
    Point p = p1.MaxNorme (p2) ;  // ou p2.maxNorme(p1)
    System.out.print ("Max de p1 et p2 : ") ; p.affiche() ;
  }
}
```

39 Récursivité des méthodes

Écrire une méthode statique d'une classe statique *Util* calculant la valeur de la "fonction d'Ackermann" *A* définie pour *m*>=0 et *n*>=0 *par :*

- A (m, n) = A (m-1, A(m, n-1)) pour *m*>0 et *n*>0,
- A (0, n) = n+1 pour *n*>0,
- A (m, 0) = A(m-1, 1) pour *m*>0.

Solution

Il suffit d'exploiter les possibilités de récursivité de Java en écrivant quasi textuellement les définitions récursives de la fonction *A*.

```
class Util
{ public static int acker (int m, int n)
  { if ( (m<0) || (n<0) ) return 0 ;  // protection : 0 si arguments incorrects
    else if (m == 0) return n+1 ;
```

```
    else if (n == 0) return acker (m-1, 1) ;
    else return acker (m-1, acker(m, n-1)) ;
  }
}

public class Acker
{ public static void main (String args[])
  { int m, n ;
    System.out.print ("Premier parametre : ") ;
    m = Clavier.lireInt() ;
    System.out.print ("Second parametre  : ") ;
    n = Clavier.lireInt() ;
    System.out.println ("acker (" + m + ", " + n + ") = " + Util.acker(m, n)) ;
  }
}
```

40 Mode de transmission des arguments d'une méthode

Quels résultats fournit ce programme ?

```
class A
{ public A (int nn)
  {n = nn ;
  }
  public int getn ()
  { return n ;
  }
  public void setn (int nn)
  { n = nn ;
  }
  private int n ;
}

class Util
{ public static void incre (A a, int p)
  { a.setn (a.getn()+p);
  }
  public static void incre (int n, int p)
  { n += p ;
  }
}
```

```
public class Trans
{ public static void main (String args[])
  { A a = new A(2) ;
    int n = 2 ;
    System.out.println ("valeur de a avant : " + a.getn()) ;
    Util.incre (a, 5) ;
    System.out.println ("valeur de a apres : " + a.getn()) ;
    System.out.println ("valeur de n avant : " + n) ;
    Util.incre (n, 5) ;
    System.out.println ("valeur de n apres : " + n) ;
  }
}
```

Solution

En Java, le transfert des arguments à une méthode se fait toujours par valeur. Mais la valeur d'une variable de type objet est sa référence. D'où les résultats :

```
valeur de a avant : 2
valeur de a apres : 7
valeur de n avant : 2
valeur de n apres : 2
```

41 Objets membres

On dispose de la classe *Point* suivante permettant de manipuler des points d'un plan.

```
class Point
{ public Point (double x, double y)          { this.x = x ;  this.y = y ; }
  public void deplace (double dx, double dy) { x += dx ;     y += dy ;    }
  public void affiche ()
  { System.out.println ("coordonnees = " + x + " " + y ) ;
  }
  private double x, y ;
}
```

En ajoutant les fonctionnalités nécessaires à la classe *Point*, réaliser une classe *Segment* permettant de manipuler des segments d'un plan et disposant des méthodes suivantes :

```
segment (Point origine, Point extremite)
segment (double xOr, double yOr, double xExt, double yExt)
double longueur() ;
void deplaceOrigine (double dx, double dy)
void deplaceExtremite (double dx, double dy)
void affiche()
```

Solution

Pour l'instant, la classe *Point* n'est dotée ni de méthodes d'accès aux champs *x* et *y*, ni de méthodes d'altération de leurs valeurs.

Si l'on prévoit de représenter un segment par deux objets de type *Point*[1], il faudra manifestement pouvoir connaître et modifier leurs coordonnées pour pouvoir déplacer l'origine ou l'extrémité du segment. Pour ce faire, on pourra par exemple ajouter à la classe *Point* les quatre méthodes suivantes :

```
public double getX ()
{ return x ;
}
public double getY ()
{ return y ;
}
public void setX (double x)
{ this.x = x ;
}
public void setY (double y)
{ this.y = y ;
}
```

En ce qui concerne la méthode *affiche* de *Segment*, on peut se contenter de faire appel à celle de *Point*, pour peu qu'on se contente de la forme du message qu'elle fournit.

Voici la nouvelle définition de *Point* et celle de *Segment :*

```
class Point
{ public Point (double x, double y)          { this.x = x ;  this.y = y ; }
  public void deplace (double dx, double dy) { x += dx ;     y += dy ;    }
  public double getX () { return x ; }
  public double getY () { return y ; }
  public void setX (double x) { this.x = x ; }
  public void setY (double y) { this.y = y ; }
  public void affiche ()
  { System.out.println ("coordonnees = " + x + " " + y ) ;
  }
  private double x, y ;
}

class Segment
{ public Segment (Point or, Point ext)
  { this.or = or ; this.ext = ext ;
  }
  public Segment (double xOr, double yOr, double xExt, double yExt)
  { or =  new Point (xOr, yOr) ;
    ext = new Point (xExt, yExt) ;
  }
```

1. On pourrait se contenter d'ajouter des méthodes *getX* et *getY*, en représentant un segment, non plus par deux points, mais par quatre valeurs de type *double*, ce qui serait moins commode.

```
  public double longueur()
  { double xOr =  or.getX(),  yOr =  or.getY() ;
    double xExt = ext.getX(), yExt = ext.getY() ;
    return Math.sqrt ( (xExt-xOr)*(xExt-xOr) + (yExt-yOr)*(yExt-yOr) ) ;
  }
  public void deplaceOrigine (double dx, double dy)
  { or.setX (or.getX() + dx) ;
    or.setY (or.getY() + dy) ;
  }
  public void deplaceExtremite (double dx, double dy)
  { ext.setX (ext.getX() + dx) ;
    ext.setY (ext.getY() + dy) ;
  }
  public void affiche ()
  { System.out.print ("Origine -   ") ; or.affiche() ;
    System.out.print ("Extremite - ") ; ext.affiche() ;
  }
  private Point or, ext ;
}
```

Voici un petit programme de test, accompagné de son résultat

```
public class TstSeg
{ public static void main (String args[])
  { Point a = new Point(1, 3) ;
    Point b = new Point(4, 8) ;
    a.affiche() ; b.affiche() ;

    Segment s1 = new Segment (a, b) ;
    s1.affiche() ;
    s1.deplaceOrigine (2, 5) ;
    s1.affiche() ;

    Segment s2 = new Segment (3, 4, 5, 6) ;
    s2.affiche() ;
    System.out.println ("longueur = " + s2.longueur()) ;
    s2.deplaceExtremite (-2, -2) ;
    s2.affiche() ;
  }
}
```

```
coordonnees = 1.0 3.0
coordonnees = 4.0 8.0
Origine -   coordonnees = 1.0 3.0
Extremite - coordonnees = 4.0 8.0
Origine -   coordonnees = 3.0 8.0
Extremite - coordonnees = 4.0 8.0
Origine -   coordonnees = 3.0 4.0
Extremite - coordonnees = 5.0 6.0
longueur = 2.8284271247461903
Origine -   coordonnees = 3.0 4.0
Extremite - coordonnees = 3.0 4.0
```

42 Synthèse : repères cartésiens et polaires

Soit la classe *Point* ainsi définie :

```
class Point
{ public Point (double x, double y)          { this.x = x ; this.y = y ; }
  public void deplace (double dx, double dy) { x += dx ;  y += dy ; }
  public double abscisse () { return x ; }
  public double ordonnee () { return y ; }
  private double x ;     // abscisse
  private double y ;     // ordonnee
}
```

La compléter en la dotant des méthodes suivantes :

- *homothetie* qui multiplie les coordonnées par une valeur (de type *double*) fournie en argument,
- *rotation* qui effectue une rotation dont l'angle est fourni en argument,
- *rho* et *theta* qui fournissent les coordonnées polaires du point,
- *afficheCart* qui affiche les coordonnées cartésiennes du point,
- *affichePol* qui affiche les coordonnées polaires du point.

Solution

La méthode *homothetie* ne présente aucune difficulté. En revanche, la méthode *rotation* nécessite une transformation intermédiaire des coordonnées cartésiennes du point en coordonnées polaires. De même, les méthode *rho* et *theta* doivent calculer respectivement le rayon vecteur et l'angle d'un point à partir de ses coordonnées cartésiennes.

Le calcul du rayon vecteur étant simple, nous l'avons laissé figurer dans les méthodes concernées (*rotation*, *rho* et *affichePol*). En revanche, le calcul d'angle a été réalisé par une méthode de service statique privée nommée *angle*. Nous y utilisons la méthode *Math.atan2* (qui reçoit en argument une abscisse *xx* et une ordonnée *yy*) plus pratique que *atan* (à laquelle il faudrait fournir le quotient *yy/xx*) car elle évite d'avoir à s'assurer que *xx* n'est pas nulle. Le résultat est un angle compris dans l'intervalle [-pi/2, pi/2] que l'on adapte en fonction des signes effectifs de *xx* et de *yy*.

Voici la définition de notre classe *Point* :

```
class Point
{ public Point (double x, double y)          { this.x = x ; this.y = y ; }
  public void deplace (double dx, double dy) { x += dx ;  y += dy ; }
  public double abscisse () { return x ; }
  public double ordonnee () { return y ; }
  public void homothetie (double coef) { x *= coef ; y *= coef ; }
```

```
  public void rotation (double th)
  { double r = Math.sqrt (x*x + y*y) ;
    double t = angle (x, y) ;
    t += th ;
    x = r * Math.cos(t) ;
    y = r = Math.sin(t) ;
  }
  public double rho()    { return Math.sqrt (x*x + y*y) ; }
  public double theta () { return angle (x, y) ; }
  public void afficheCart ()
  { System.out.println ("Coordonnees cartesiennes = " + x + " " + y ) ;
  }
  public void affichePol ()
  { System.out.println ("Coordonnees polaires = " + Math.sqrt (x*x + y*y)
                                                  + " " + angle (x, y) ) ;
  }
  private static double angle (double xx, double yy)
           // methode de service (on choisit une determination
           //  de l'angle entre -pi et +pi)
  { double a = Math.atan2 (yy, xx) ;
    if (yy<0) if (xx>=0) return a + Math.PI ;
                  else return a - Math.PI ;
    return a ;
  }
  private double x ;     // abscisse
  private double y ;     // ordonnee
}
```

Voici à titre indicatif un petit programme d'essai, accompagné du résultat de son exécution :

```
public class PntPol
{ public static void main (String args[])
  { Point a ;
    a = new Point(1, 1) ;       a.afficheCart() ; a.affichePol() ;
    a.deplace(-1, -1) ;         a.afficheCart() ; a.affichePol() ;
    Point b = new Point(1, 0) ; b.afficheCart() ; b.affichePol() ;
    b.homothetie (2) ;          b.afficheCart() ; b.affichePol() ;
    b.rotation (Math.PI) ;      b.afficheCart() ; b.affichePol() ;
  }
}
```

```
Coordonnees cartesiennes = 1.0 1.0
Coordonnees polaires = 1.4142135623730951 0.7853981633974483
Coordonnees cartesiennes = 0.0 0.0
Coordonnees polaires = 0.0 0.0
Coordonnees cartesiennes = 1.0 0.0
Coordonnees polaires = 1.0 0.0
Coordonnees cartesiennes = 2.0 0.0
Coordonnees polaires = 2.0 0.0
Coordonnees cartesiennes = -2.0 1.2246467991473532E-16
Coordonnees polaires = 2.0 3.141592653589793
```

43 Synthèse : modification de l'implémentation d'une classe

Modifier la classe *Point* réalisée dans l'exercice 42, de manière que les données (privées) soient maintenant les coordonnées polaires d'un point et non plus ses coordonnées cartésiennes. On fera en sorte que le "contrat" initial de la classe soit respecté en évitant de modifier les champs publics ou les en-têtes de méthodes publiques (l'utilisation de la classe devra continuer à se faire de la même manière).

Solution

Le constructeur reçoit toujours en argument les coordonnées cartésiennes d'un point. Il doit donc opérer les transformations appropriées.

Par ailleurs, la méthode *deplace* reçoit un déplacement exprimé en coordonnées cartésiennes. Il faut donc tout d'abord déterminer les coordonnées cartésiennes du point après déplacement, avant de repasser en coordonnées polaires.

En revanche, les méthodes *homothetie* et *rotation* s'expriment maintenant très simplement.

Voici la définition de notre nouvelle classe. Nous faisons appel à la même méthode de service *angle* que dans l'exercice précédent.

```
class Point
{ public Point (double x, double y)
  { rho = Math.sqrt (x*x + y*y) ;
    theta = Math.atan (y/x) ;
  }
  public void deplace (double dx, double dy)
  { double x = rho * Math.cos(theta) + dx ;
    double y = rho * Math.sin(theta) + dy ;
    rho = Math.sqrt (x*x + y*y) ;
    theta = angle (x, y) ;
  }
  public double abscisse () { return rho * Math.cos(theta) ; }
  public double ordonnee () { return rho * Math.sin(theta) ; }
  public void homothetie (double coef) { rho *= coef ; }
  public void rotation (double th)
  { theta += th ;
  }
  public double rho()    { return rho ; }
  public double theta () { return theta ; }
  public void afficheCart ()
  { System.out.println ("Coordonnees cartesiennes = " + rho*Math.cos(theta)
                        + " " + rho*Math.sin(theta) ) ;
  }
```

```
  public void affichePol ()
  { System.out.println ("Coordonnees polaires = " + rho + " " + theta) ;
  }
  private static double angle (double xx, double yy)
          // methode de service (on choisit une determination
          //  de l'angle entre -pi et +pi)
  { double a = Math.atan2 (yy, xx) ;
    if (yy<0) if (xx>=0) return a + Math.PI ;
                    else return a - Math.PI ;
    return a ;
  }
  private double rho ;     // rayon vecteur
  private double theta ;   // angle polaire
}
```

À titre indicatif, nous pouvons tester notre classe avec le même programme que dans l'exercice précédent. Il fournit les mêmes résultats, aux incertitudes de calcul près :

```
public class PntPol2
{ public static void main (String args[])
  { Point a ;
    a = new Point(1, 1) ;       a.afficheCart() ; a.affichePol() ;
    a.deplace(-1, -1) ;         a.afficheCart() ; a.affichePol() ;
    Point b = new Point(1, 0) ; b.afficheCart() ; b.affichePol() ;
    b.homothetie (2) ;          b.afficheCart() ; b.affichePol() ;
    b.rotation (Math.PI) ;      b.afficheCart() ; b.affichePol() ;
  }
}
```

```
Coordonnees cartesiennes = 1.0000000000000002 1.0
Coordonnees polaires = 1.4142135623730951 0.7853981633974483
Coordonnees cartesiennes = 2.220446049250313E-16 0.0
Coordonnees polaires = 2.220446049250313E-16 0.0
Coordonnees cartesiennes = 1.0 0.0
Coordonnees polaires = 1.0 0.0
Coordonnees cartesiennes = 2.0 0.0
Coordonnees polaires = 2.0 0.0
Coordonnees cartesiennes = -2.0 2.4492127076447545E-16
Coordonnees polaires = 2.0 3.141592653589793
```

44 Synthèse : vecteurs à trois composantes

Réaliser une classe *Vecteur3d* permettant de manipuler des vecteurs à trois composantes (de type *double*) et disposant :

- d'un constructeur à trois arguments,
- d'une méthode d'affichage des coordonnées du vecteur, sous la forme :

  < composante_1, composante_2, composante_3 >

- d'une méthode fournissant la norme d'un vecteur,
- d'une méthode (statique) fournissant la somme de deux vecteurs,
- d'une méthode (non statique) fournissant le produit scalaire de deux vecteurs.

Écrire un petit programme (*main*) utilisant cette classe.

Solution

```
class Vecteur3d
{ public Vecteur3d (double x, double y, double z)
  { this.x = x ; this.y = y ; this.z = z ;
  }
  public void affiche ()
  { System.out.println ("< " + x + ", " + y + ", " + z + " >") ;
  }
  public double norme ()
  { return (Math.sqrt (x*x + y*y + z*z)) ;
  }
  public static Vecteur3d somme(Vecteur3d v, Vecteur3d w)
  { Vecteur3d s = new Vecteur3d (0, 0, 0) ;
    s.x = v.x + w.x ; s.y = v.y + w.y ; s.z = v.z + w.z ;
    return s ;
  }
  public double pScal (Vecteur3d v)
  { return (x*v.x + y*v.y + z*v.z) ;
  }
  private double x, y, z ;
}
public class TstV3d
{ public static void main (String args[])
  { Vecteur3d v1 = new Vecteur3d (3, 2, 5) ;
    Vecteur3d v2 = new Vecteur3d (1, 2, 3) ;
    Vecteur3d v3 ;
    System.out.print ("v1 = " ) ; v1.affiche() ;
    System.out.print ("v2 = " ) ; v2.affiche() ;
    v3 = Vecteur3d.somme (v1, v2) ;
    System.out.print ("v1 + v2 = " ) ; v3.affiche() ;
    System.out.println ("v1.v2 = " + v1.pScal(v2)) ;   // ou v2.pScal(v1)
  }
}
```

```
v1 = < 3.0, 2.0, 5.0 >
v2 = < 1.0, 2.0, 3.0 >
v1 + v2 = < 4.0, 4.0, 8.0 >
v1.v2 = 22.0
```

Remarque

1. Le corps de la méthode *somme* pourrait être écrit de façon plus concise :

```
return new Vecteur3d (v.x+w.x, v.y+w.y, v.z+w.z) ;
```

Remarque

2. Les instructions suivantes de *main* :

```
v3 = Vecteur3d.somme (v1, v2) ;
System.out.print ("v1 + v2 = " ) ; v3.affiche() ;
```

pourraient être remplacées par :

```
System.out.print ("v1 + v2 = " ) ; (Vecteur3d.somme(v1, v2)).affiche() ;
```

3. Si la méthode *pScal* avait été prévue statique, son utilisation deviendrait symétrique. Par exemple, au lieu de *v1.pScal(v2)* ou *v2.pScal(v1)*, on écrirait *Vecteur3d.pScal(v1, v2)*.

45 Synthèse : nombres sexagésimaux

On souhaite disposer d'une classe permettant d'effectuer des conversions (dans les deux sens) entre nombre sexagésimaux (durée exprimée en heures, minutes, secondes) et des nombres décimaux (durée exprimée en heures décimales). Pour ce faire, on réalisera une classe permettant de représenter une durée. Elle comportera :

- un constructeur recevant trois arguments de type *int* représentant une valeurs sexagésimale (heures, minutes, secondes) qu'on supposera normalisée (secondes et minutes entre 0 et 59). Aucune limitation ne portera sur les heures ;
- un constructeur recevant un argument de type *double* représentant une durée en heures ;
- une méthode *getDec* fournissant la valeur en heures décimales associée à l'objet,
- des méthodes *getH*, *getM* et *getS* fournissant les trois composantes du nombre sexagésimal associé à l'objet.

On proposera deux solutions :

1. Avec un champ (privé) représentant la valeur décimale,
2. Avec des champs (privés) représentant la valeur sexagésimale.

Solution

En conservant la valeur décimale

Les deux constructeurs ne posent pas de problème particulier, le second devant simplement calculer la durée en heures correspondant à un nombre donné d'heures, de minutes et de secondes. Les méthodes *getH*, *getM* et *getS* utilisent le même principe : le nombre d'heures n'est rien d'autre que la partie entière de la durée décimale. En le soustrayant de cette durée décimale, on obtient un résidu d'au plus une heure qu'on convertit en minutes en le multipliant par 60. Sa partie entière fournit le nombre de minutes qui, soustrait du résidu horaire fournit un résidu d'au plus une minute...

```
class SexDec
{ public SexDec (double dec)
  { this.dec = dec ;
  }
  public SexDec (int h, int mn, int s)
  { dec = h + mn/60. + s/3600. ;
  }
  public double getDec()
  { return dec ;
  }
  public int getH()
  { int h = (int)dec ; return h ;
  }
  public int getM()
  { int h = (int)dec ;
    int mn = (int)(60*(dec-h)) ;
    return mn ;
  }
  public int getS()
  { int h = (int)dec ;
    double minDec = 60*(dec-h) ;
    int mn = (int)minDec ;
    int sec = (int)(60*(minDec-mn)) ;
    return sec ;
  }
  private double dec ;
}
```

Voici un petit programme de test, accompagné du résultat d'exécution :

```
public class TSexDec1
{ public static void main (String args[])
  { SexDec h1 = new SexDec(4.51) ;
    System.out.println ("h1 - decimal = " + h1.getDec()
       +"  Sexa = " + h1.getH() + " " + h1.getM() + " " + h1.getS()) ;
   SexDec h2 = new SexDec (2, 32, 15) ;
   System.out.println ("h2 - decimal = " + h2.getDec()
       +"  Sexa = " + h2.getH() + " " + h2.getM() + " " + h2.getS()) ;
}
```

```
h1 - decimal = 4.51  Sexa = 4 30 35
h2 - decimal = 2.5375  Sexa = 2 32 15
```

En conservant la valeur sexagésimale

Cette fois, le constructeur recevant une valeur en heures décimales doit opérer des conversions analogues à celles opérées précédemment par les méthodes d'accès *getH*, *getM* et *getS*. En revanche, les autres méthodes sont très simples.

```
class SexDec
{ public SexDec (double dec)
  { h =  (int)dec ;
    int minDec = (int)(60*(dec-h)) ;
    mn = (int)minDec ;
    s = (int)(60*(minDec-mn)) ;
  }
  public SexDec (int h, int mn, int s)
  { this.h = h ; this.mn = mn ; this.s = s ;
    }
  public double getDec()
  { return (3600*h+60*mn+s)/3600. ;
  }
  public int getH()
  { return h ;
  }
  public int getM()
  { return mn ;
  }
  public int getS()
  { return s ;
  }
  private int h, mn, s ;
}
```

Voici le même programme de test que précédemment, accompagné de son exécution :

```
public class TSexDec2
{ public static void main (String args[])
  { SexDec h1 = new SexDec(4.51) ;
    System.out.println ("h1 - decimal = " + h1.getDec()
       +"  Sexa = " + h1.getH() + " " + h1.getM() + " " + h1.getS()) ;
   SexDec h2 = new SexDec (2, 32, 15) ;
   System.out.println ("h2 - decimal = " + h2.getDec()
       +"  Sexa = " + h2.getH() + " " + h2.getM() + " " + h2.getS()) ;
  }
}
```

```
h1 - decimal = 4.5  Sexa = 4 30 0
h2 - decimal = 2.5375  Sexa = 2 32 15
```

Remarque

On notera que la première démarche permet de conserver une durée décimale atteignant la précision du type *double*, quitte à ce que la valeur sexagésimale correspondante soit arrondie à la seconde la plus proche. La deuxième démarche, en revanche, en imposant d'emblée un nombre entier de secondes, entraîne une erreur d'arrondi définitive (entre 0 et 1 seconde) dès la création de l'objet. Bien entendu, on pourrait régler le problème en conservant un nombre de secondes décimal ou encore, en gérant un résidu de secondes.