Cours de Java : Les objets et les classes
Chapitre 4

1. Introduction

Une **classe** correspond à **un type défini par l'utilisateur** dans lequel **se trouvent associées des données (variables membres, ou attributs) et des méthodes (fonctions membres) qui permettent la manipulation de ces données.**
Un **objet** est une **variable de type classe**.
Une classe sert a fabriquer des objets, on peut alors imaginer qu'une classe est un moule.

2. Premier exemple : définition et utilisation d'une classe Point

On considère ici une classe nommée Point qui permet de manipuler des points situés dans un plan.
Un point est caractérisé par ses coordonnées cartésiennes (abscisse et ordonnée) dans un repère orthonormé. On peut considérer la classe Point comme une sorte de classe métier.

2.1 Définition de la classe Point

```
public class Point {

        public double x;        //abscisse
        public double y;        //ordonnée

        public void place(double xnew, double ynew)
        {
                x = xnew;       //on affecte dans l'attribut x la valeur reçue en 1er paramètre
                y = ynew;
        }

        public void deplace(double dx, double dy)
        {
                x += dx;  //on ajoute à l'attribut x la valeur reçue en 1er paramètre
                y += dy;
        }

        public void affiche()
        {
                System.out.println("x = " + x + ", y = "  + y );
        }
}
```

Elle est placée dans un fichier nommé Point.java.
La classe Point correspond à un nouveau **type** de données. Elle possède :
- **2 attributs** (ou variables membres), tous les 2 de type double et nommés x et y qui correspondent aux coordonnées d'un point (abscisse et ordonnée).
- **3 méthodes** ou fonctions membres :
    - *place()* qui sert à attribuer des valeurs aux coordonnées d'un point;
    - *deplace()* qui sert à modifier les coordonnées d'un point;
    - *affiche()* qui permet d'afficher les coordonnées d'un point.

Remarques :
- le mot-clé **`public`** indique le **niveau de visibilité ou d'accessibilité** des membres (attributs ou méthodes). **`public`** signifie accessible partout, y compris à l'"extérieur" de la classe.

2.2. Utilisation de la classe Point

L'utilisation de la classe **Point** se fait dans une autre classe nommée **TestPoint** à créer dans le même dossier que la classe **Point** ou dans le même package que la classe **Point** (si par exemple on travaille avec Eclipse).
La définition de cette classe **TestPoint** (placée dans le fichier TestPoint.java) contient la méthode **main()** qui constitue le point d'entrée de l'exécution de l'application.
La classe **TestPoint** sert, comme son nom l'indique, à tester la classe "métier" Point.

```
public class TestPoint {

        public static void main(String [] args)  {
                Point a, b;                     // déclaration de 2 objets nommés a et b  (*1)
                a = new Point();                // création d'un objet de la classe Point    (*2)
                b = new Point();                // création d'un objet de la classe Point    (*2)
                a.x = 6.3;                      //(*3)
                a.y = 8.5;                      //(*4)
                a.affiche();                    //(*5)
                a.deplace(1.5, 2.6);            //(*6)
                a.affiche();
                b.place(1.1, 2.2);
                b.affiche();
        }
}
```

(*1) : la déclaration des objets correspond plus exactement à la **déclaration de 2 variables références** nommée a et b destinées à contenir l'adresse mémoire (ou la référence) d'un objet de type classe Point.

(*2) : l'opérateur **new permet de créer un objet, ce qui implique la réservation de l'espace mémoire occupé par cet objet.** Cet espace mémoire est effacé, les champs sont mis à 0, false ou null suivant leur type. Puis le constructeur invoqué est appelé, voir le paragraphe 4.

Remarques :

- Dans le jargon de la P.O.O. (Programmation Orientée Objet), on dit que a et b sont des **instances** de la classe Point. On dit également qu'on **instancie** la classe Point lorsqu'on **crée** un objet de type classe Point. La **création** d'un objet constitue une **instanciation** de la classe correspondante.
- lors de la génération d'un exécutable, les codes des méthodes ne sont eux générés qu'une seule fois (le contraire conduirait à un gaspillage de mémoire !).

(*3) : l'expression **`a.x`** permet d'accéder à l'attribut x de l'objet a. Ici, on y affecte la valeur 6.3.
(*4) : l'expression **`a.y`** permet d'accéder à l'attribut y de l'objet a. Ici, on y affecte la valeur 8.5.

Remarque : pour **accéder à un membre** (variable ou fonction) d'un objet à l'extérieur de sa classe, il faut utiliser le nom de l'objet (en réalité une référence sur cet objet) suivi de l'**opérateur "."**, suivi du nom du membre.
Chaque ligne suivante correspond à **un appel d'une méthode pour un objet**.
(*5) : on **appelle la méthode *affiche()* pour l'objet a**. Ici, on souhaite afficher les coordonnées du point a (c'est à dire les valeurs de ses attributs x et y). Le **préfixe *a.*** permet de **préciser à la fonction**

membre quel est l'objet sur lequel elle doit opérer : on dit que l'objet a constitue le **paramètre implicite transmis à la méthode** lors de son appel (il est transmis par référence).

- L'appel de la méthode *affiche()* pour un objet ne nécessite pas la transmission de paramètre supplémentaire (chaque objet "contient" ses attributs).

(*6): on appelle la fonction *deplace()* pour l'objet a en lui transmettant en paramètres les valeurs 1.5 et 2.6.

- en jargon P.O.O., on dit également que `a.deplace(1.5, 2.6);` constitue **l'envoi d'un message** (*deplace,* accompagné des informations 1.5 et 2.6) à l'objet a.
- ici, la méthode *deplace()* est **capable de modifier l'objet** (c'est à dire ses attributs) **pour lequel elle est appelée**. Par défaut, c'est le cas de toutes les méthodes (normal car le paramètre implicite est transmis par référence!).
- dans le corps de chaque méthode, l'expression **x** permet **d'accéder à l'attribut x de l'objet pour lequel la méthode a été appelée**. Comme ces 3 méthodes ne sont pas statiques, **on ne peut que les appeler pour un objet créé auparavant**.

- Pour compiler les 2 fichiers sources :
  javac Point.java
  javac TestPoint.java
- Pour exécuter le programme :
  java TestPoint

Q1 Tester ce 1er exemple

- Résultat de l'exécution :

```
x = 6.3, y = 8.5
x = 7.8, y = 11.1
x = 1.1, y = 2.2
```

Remarque :
- Il est possible, dans la définition d'une méthode, d'appeler une autre méthode (pour l'objet pour lequel la 1ère méthode a été appelée). Exemple : nouvelle définition de la méthode *place()* :

```
public void place(double ix, double iy)
{
        x = ix;
        y = iy;
        // appel de la méthode deplace() pour l'objet pour lequel
        // la méthode place() a été appelée
        deplace(0, 0); // sans effet !
}
```

3. Encapsulation des données

Dans un objet de type classe, on trouve des variables membres (attributs) et des fonctions membres (méthodes). Mais cette association ne suffit pas à réaliser une **programmation orientée objet** (POO). En effet, dans ce qu'on pourrait qualifier de POO « pure », on réalise ce que l'on nomme une **encapsulation des données.** Cela signifie que les utilisateurs d'une classe (la classe TestPoint dans l'exemple précédent) **ne peuvent pas agir directement sur les données d'une instance de cette classe** : **les données d'un objet ne sont pas directement accessibles à l'extérieur de la classe.** Un

utilisateur qui souhaite agir sur les données d'une instance d'une classe, doit alors obligatoirement se servir des méthodes de la classe, qui jouent le rôle d'**interface** obligatoire.

Le grand mérite de l'encapsulation est que, vu de l'extérieur, un objet se caractérise uniquement par les spécifications de ses méthodes, la manière dont sont réellement implantées les données est transparente pour l'utilisateur de la classe.
L'encapsulation des données présente un intérêt manifeste en matière de qualité logicielle :

- elle **sécurise l'accès aux données** contenues dans un objet ;
- elle facilite **la maintenance** : une modification éventuelle de la "structure" des données d'une classe n'a d'incidence que sur la classe elle-même ; les utilisateurs de la classe ne sont pas concernés par la teneur de cette modification ;
- elle facilite grandement la **réutilisation d'une classe**.

Pour encapsuler les données d'une classe, il est nécessaire et suffisant que tous ses **attributs soient privés.** Seules les fonctions membres d'une classe ont un accès direct aux membres privés de cette même classe. Les fonctions, qui ne sont pas membres de cette classe, ont, quant à elles, seulement un accès direct aux membres publics de la classe.

3.1 Encapsulation des données de la classe Point : nouvelle définition de la classe Point

Pour encapsuler les données de la classe Point, il suffit de déclarer ses attributs de telle sorte qu'ils soient **privés**.

```
public class Point {

        private double x;       //abscisse
        private double y;       //ordonnée

        public void place(double ix, double iy)
        {
                x = ix;  //on affecte dans l'attribut x la valeur reçue en 1er paramètre
                y = iy;
        }

        public void deplace(double dx, double dy)
        {
                x += dx;  //on ajoute à l'attribut x la valeur reçue en 1er paramètre
                y += dy;
        }

        public void affiche()
        {
                System.out.println("x = " + x + ", y = "  + y );
        }
}
```

Remarques :
- les mots-clés **`private ou public`** indiquent le **niveau de visibilité ou d'accessibilité** des membres (attributs ou méthodes) :
  - **`private`** signifie accessible uniquement à l'intérieur de la classe (c'est à dire aux méthodes de la classe), et donc inaccessible à l'extérieur de la classe.
  - **`public`** signifie accessible partout, y compris à l'"extérieur" de la classe.
- il est également possible de déclarer des méthodes privées (afin de les rendre inaccessibles à l'extérieur de la classe). De telles fonctions membres ne peuvent être appelées que par d'autres fonctions membres.

Q2 Modifier la définition de la classe Point pour encapsuler ses données. Recompiler. Supprimer dans la définition de la classe TestPoint les instructions qui posent problème, et remplacer les par une instruction qui permet d'obtenir le même résultat lors de l'exécution.

La fonction main() (fonction membre de la classe TestPoint) n'a plus directement accès aux attributs privés des objets a et b. Si on souhaite obtenir le même résultat, il suffit d'appeler la méthode *place()* pour l'objet nommé a en lui transmettant en paramètres les valeurs 6.3 et 8.5.

3.2 Les accesseurs : méthodes get() et set()

Comme les utilisateurs de la classe Point n'ont pas un accès direct aux attributs d'une instance de cette classe, il **peut** s'avérer nécessaire de déclarer et définir des méthodes qui permettent d'accéder (indirectement) à ces attributs.
Pour chaque attribut :
- une méthode **getXxx()** permet d'**accéder en consultation** à la valeur de l'attribut (xxx doit correspondre au nom de l'attribut);
- une méthode **setXxx()** permet d'**accéder en modification** à la valeur de l'attribut.

3.2.1 Définition de méthodes get() et set() pour la classe Point

On ajoute les méthodes suivantes dans la définition de la classe Point

```
public void setX(double newx)  { // accès en modification à l'attribut x
        x = newx; }

public double getX()    {        // accès en consultation à l'attribut x
        return x; }

public void setY(double newy)  {
        y = newy; }

public double getY()  {
        return y; }
```

3.2.2 Exemple d'utilisation (dans la méthode main() de la classe TestPoint)

```
Point a;
a = new Point();
a.setX(8.7);
System.out.println(a.getX());
```

Remarque : l'expression **`a.getX()`** est du type de la valeur de retour de la méthode **`getX()`**, donc du type `double`.

Q3 Tester cet exemple

4. Constructeur d'une classe

Il s'agit d'une **fonction membre qui a le même nom que la classe**.
Il peut comporter un nombre quelconque de paramètres, éventuellement aucun. Par définition, un **constructeur ne renvoie pas de valeur** : **aucun type ne doit figurer devant son nom** (dans ce cas précis, la présence de void est une erreur).

En théorie, un constructeur peut être public ou privé. En pratique, à moins d'avoir de bonnes raisons de faire le contraire, il **vaut mieux le rendre public** (car si il est privé, il n'est plus accessible à l'extérieur de la classe !).
Il est **appelé automatiquement lorsque l'on instancie la classe** (c'est à dire lorsqu'on **crée un objet de cette classe avec l'opérateur new**).

Son rôle : réaliser toutes actions à effectuer avant de pouvoir utiliser "convenablement" l'objet créé. Pour les classes les plus simples, **le constructeur sert notamment à donner une valeur initiale à chaque attribut**. Pour des classes plus compliquées, le constructeur peut également être utilisé pour vérifier l'existence ou ouvrir un fichier, se connecter à une base de données ...

Attention :
- si on ne modifie pas la définition de la classe Point, le code suivant exécuté dans la fonction main() de la classe TestPoint ne pose pas de problème.

```
Point a;
a = new Point();
a.affiche(); // affiche : x = 0.0, y = 0.0
```

Pourquoi ?
Réponse : Lorsqu'une classe ne définit aucun constructeur, un constructeur par défaut est automatiquement créé par la JVM.
On peut voir ce constructeur par défaut avec la définition suivante :

```
public Point() {}
```

Le constructeur par défaut créé par la JVM ne reçoit pas d'arguments et n'exécute pas d'instruction.

4.1 Définition d'un constructeur pour la classe Point (version 1)

On ajoute un constructeur avec 2 paramètres dans la définition de la classe Point :

```
public class Point {
        private double x ;
        private double y ;
        public Point(double ix, double iy)      //i pour valeur initiale
        {       x = ix;
                y = iy;
                //pour debug et pour voir
                System.out.println("Constructeur avec arguments,  x = " + x+" y = "+y);
        }
        //.....
}
```

Ici, le constructeur possède 2 paramètres qui sont destinées à attribuer des valeurs initiales aux coordonnées du point. Le constructeur est publique afin qu'ils soit accessible à l'extérieur de la classe Point.
Remarque : ici, l'affichage sert seulement à "tracer" les appels du constructeur.

Modification de la classe TestPoint :

```
public class TestPoint {
        public static void main(String [] args) {
                Point a, b;
                a = new Point( 5 , 8 );
                b = new Point( 4.2 , 7.5 );
                a.affiche();
        }
}
```

Q4 Tester cet exemple.

Remarque : - ici, une instruction telle que `a = new Point();` est rejetée par le compilateur car **à partir du moment où une classe possède un constructeur, il n'est plus possible d'instancier cette classe avec le constructeur par défaut. Et ici pour appeler le constructeur, il faut impérativement lui transmettre 2 paramètres de type double.**

4.2 Définition d'un constructeur pour la classe Point (version 2)

On modifie la définition du constructeur afin qu'il soit sans paramètre :

```
public class Point {
        private double x ;
        private double y ;
        public Point()
        {
                x = 0.0;  y = 0.0;
        }
}
```

Remarque : il est préférable (pour la lisibilité du code) qu'un constructeur, même sans paramètre, initialise (à des valeurs par défaut) tous les attributs de l'objet, même si dans ce cas précis ça ne sert à rien !

Q5 Modifier la classe TestPoint pour appeler ce constructeur.

4.3 Initialisation des attributs lors de leur déclaration

Le langage Java offre cette possibilité. Exemple (à ne pas reproduire !) :

```
public class Point {
        private double x  = 8;
        private double y  = 9;
        public Point()
        {
                x = 0.0;
                System.out.println("Appel du constructeur de la classe Point");
        }
}
```

```
public class TestPoint {
        public static void main(String [] args)
        {
                Point a, b;
                a = new Point(); b = new Point();
                a.affiche(); b.affiche();
        }
}
```

Q6 Tester ce code.

Conclusion : c'est l'initialisation dans le constructeur qui gagne, mais question lisibilité, on peut faire mieux !

Remarque : **en Java, quoique vous fassiez, les attributs d'un objet seront toujours initialisés !**

Un constructeur est une méthode particulière qui porte le même nom que sa classe, qui est sans type, pas même void, et qui peut éventuellement recevoir des arguments.

5. Surcharge de méthode

D'une manière générale, on parle de "surcharge" lorsqu'un même symbole (ici le nom de la fonction) possède plusieurs significations différentes. Surcharger un nom de fonction revient à définir plusieurs fonctions qui portent ce même nom.
Pour pouvoir employer plusieurs fonctions de même nom, il faut un critère (autre que le nom) permettant de **choisir la bonne fonction lors d'un appel**. En Java, ce choix **est basé sur le type et le nombre des paramètres**.
Toutes les méthodes d'une classe, y compris le constructeur, peuvent être surchargées.

Exemple :

```
public class Point {
        private double x;
        private double y;

        public Point(double ix, double iy)      {
                System.out.println("Appel du constructeur 2 de la classe Point");
                x = ix; y = iy;
        }

        public Point(double i)  {
                System.out.println("Appel du constructeur 1 de la classe Point");
                x = i; y = i;
        }

        public Point()  {
                System.out.println("Appel du constructeur 0 de la classe Point");
                x = 0.0; y = 0.0;
        }

        public void place(double xnew, double ynew)  {
                x = xnew;
                y = ynew;
        }

        public void place(int val) {
                x = val;            //conversion non dégradante !
                y = val;
        }
        public void affiche()    {
                System.out.println("x = " + x + ", y = "  + y );
        }
}
```

Le constructeur est surchargé : il en existe 3 versions différentes.
De même, la méthode place() est surchargée: il en existe 2 versions.

```
public class TestPoint {
        public static void main(String [] args) {
                Point a, b;
                a = new Point();
                b = new Point(1);
```

```
                b = new Point(2,2.0);
                // à cet instant là, le 1er point b créé n'est plus accessible !
                b.affiche();
                a.place(3, 4.0);
                a.place(5.5);     // ne compile pas !
                a.affiche();
        }
}
```

L'instruction a.place(5.5) ne compile pas car au moment de l'appel de cette méthode, le système crée une variable pour le paramètre formel nommé val (cf. définition de la méthode place(int val)) et cherche à l'initialiser avec une expression constante de type double : il doit donc convertir un double en int, ce qui constitue une conversion dégradante !

Q7 Corriger ce problème, afin de pouvoir tester ce code.

6. Le mot clé this

On peut utiliser le mot clé this dans le corps d'une méthode pour **désigner l'objet pour lequel la méthode est appelé**.

Exemple 1 : Modification de la méthode place() avec 2 paramètres : sans effet, et sans grand intérêt !

```
        public void place(double xnew, double ynew)  {
                this.x = xnew;
                y = ynew;
        }
```

L'expression **this.x,** à l'intérieur de la définition de la méthode, permet d'accéder à l'attribut x de l'objet pour lequel la méthode est appelé.
On peut considérer que **this** correspond à une référence sur l'objet pour lequel la méthode est appelé.

Q8 Tester cette modification de la méthode place() avec 2 paramètres

Exemple 2 : autre modification de la méthode place() avec 2 paramètres

```
        public void place(double x, double y)   {
                System.out.println("Valeur de x : " + x); //(*1)
                x = x;
                y = y;
        }
```

Pour tester cette modification, utiliser la classe TestPoint suivante :

```
public class TestPoint {
        public static void main(String [] args) {
                Point a = new Point();
                a.place(3, 4.0);
                a.affiche();
        }
}
```

Q9 Tester cette modification de la méthode place() avec 2 paramètres

Conclusion :

(*1) : comme les noms des paramètres sont identiques aux noms des attributs, les attributs sont **masqués** : ils ne sont plus directement accessibles
Solution pour accéder aux attributs de l'objet dans ce cas : utiliser le mot clé **this**.

Q10 Tester le code suivant pour la méthode place() avec 2 paramètres

```
public void place(double x, double y)  {
        System.out.println("Valeur de x : " + x);
        this.x = x;
        this.y = y;
}
```

7. Exercice de révision : pour celles et ceux qui n'ont pas bien compris le concept de classe

7.1 Qu'est-ce qu'une classe ?

Il n'existe pas de réelle définition d'une classe en programmation objet.
Une classe sert à fabriquer des objets, on peut alors imaginer qu'une classe est un moule.

Un objet est caractérisé par un certain nombre de propriétés.
Par exemple, pour un objet rectangle on aurait les propriétés suivantes :

- largeur
- longueur
- Position du centre (x,y) dans le plan.
- angle de rotation (si le rectangle n'est pas en position horizontale).

Cet objet rectangle peut être inanimé (il ne bouge plus dans le plan) mais, en informatique, on peut lui faire subir des opérations comme :

- translater
- tourner
- agrandir
- réduire
- afficher

On définit ainsi une classe Java pour tous nos objets de type Rectangle à créer.

Ecriture simplifiée en Java de la classe **Rectangle**.

```
public class Rectangle {
                /* Les propriétés = attributs = données membres */
        float largeur ;
        float longueur ;
        //Position du centre du rectangle (X,Y) dans le plan.
        int x;
        int y ;
        float angle ;
                /* Les opérations = méthodes = fonctions membres */
        void translater(int dx, int dy)  { /* A coder */ }
        void tourner(float a)  { /* A coder */ }
        void agrandir(float a)  { /* A coder */ }
```

```
                void reduire (float a)  { /* A coder */ }
                void afficher()  { /* A coder */ }
                }
```

Les attributs sont des variables particulières car ils sont déclarés dans la classe mais à l'extérieur de toutes les méthodes. C'est pour cela qu'on dit qu'ils sont les propriétés de la classe.
Les attributs sont accessibles par toutes les méthodes de la classe.

On ne code ici que les méthodes **translater()** et **afficher()**.

```
void translater(int dx, intd y)  {
        x = x + dx ;
        y = y + dy ;
 }
void afficher()  {
        System.out.println("Ma longueur= "+longueur+ " et ma largeur= "+largeur) ;
        System.out.println("Je suis en X = "+x+ " et en Y = "+y) ;
}
```

Les méthodes afficher() et translater() accèdent aux attributs x et y.

7.2 La protection des attributs avec l'encapsulation

Elle consiste à déclarer **private** tous les attributs.

- Les champs **private** sont accessibles uniquement par les méthodes de la classe,
- Les champs précisés **public** sont accessibles à tous les autres objets qui manipulent un objet instance de la classe concernée. On dit qu'ils sont accessibles à l'extérieur de la classe.

Les méthodes peuvent être déclarées **public**.

```
public class Rectangle {
                        /* Les propriétés = attributs = données membres */
                private float largeur ;
                private float longueur ;
                //Position du centre du rectangle (X,Y) dans le plan.
                private int x ;
                private int y ;
                private  float angle ;
                        /* Les opérations = méthodes = fonctions membres */
                public void translater(int dx, int dy)  { /* A coder */ }
                public void tourner(float a)  { /* A coder */ }
                public void agrandir(float a)  { /* A coder */ }
                public void reduire (float a)  { /* A coder */ }
                public void afficher()  { /* A coder */ }
                }
```

7.3 Créer (instancier) les objets

La classe Rectangle est le moule utilisé pour fabriquer des objets de type rectangle.
Ces objets sont des objets informatiques.

Il faut maintenant créer des objets de type Rectangle à partir de la classe précédente.
Cela peut se faire en 2 étapes :

1) Déclarer une variable de type Rectangle : c'est notre objet, il s'appelle ici monrec.

```
Rectangle  monrec;
```

2) Créer l'objet avec l'opérateur **new** :

```
monrec = new Rectangle() ;
```

Où créer les objets ?

La méthode main vue dans les chapitres précédents doit être utilisée pour créer (instancier) des objets de type Rectangle.
On crée pour cela une classe d'application **MainRectangle** qui est ici très simple car elle ne contient que la méthode main().
On suppose travailler dans un même dossier, c'est à dire que les fichiers Rectangle.java et MainRectangle.java sont placés dans un même dossier ainsi que les fichiers .class correspondants..

```
public class MainRectangle {
public static  void main(String[] args)    {
        Rectangle  monrec;
        monrec = new Rectangle() ;
    }
 }
```

Comment utiliser les méthodes (fonctions membres) de la classe Rectangle ?

Par exemple, le nom de l'objet (monrec) suivi d'un point (.) suivi du nom de la méthode (afficher). On ajoute alors les 3 instructions écrites en gras.

```
public class MainRectangle {
public static  void main(String[] args)   {
        Rectangle  monrec;
        monrec = new Rectangle() ;
        monrec.afficher() ;
        monrec. translater(10,15) ;
        monrec.afficher() ;
    }
 }
```

Exercice :
Ecrire les classes Rectangle.java et MainRectangle.java dans un même dossier.
Compiler les 2 classes.
Exécuter MainRectangle.
Que valent les attributs de l'objet Rectangle créé?

Réponse : l'objet monrec est bien créé et tous ses attributs sont mis à 0...

7.4 Le constructeur de la classe Rectangle

La JVM crée un constructeur par défaut, l'exemple précédent montre que l'objet monrec est bien créé et que tous ses attributs sont mis à 0...
Cet exemple montre bien l'utilité d'un constructeur. On pourra alors choisir les dimensions du rectangle créé, sa place dans le plan et son orientation par défaut.

```
public class Rectangle {
                    private float largeur ;
                    private float longueur ;
                    //Position du centre (X,Y) dans le plan.
                    private int x ;
                    private int x ;
                    private float angle ;
 public Rectangle() {
        largeur = 10 ;
        longueur = 20 ;
        x= 10 ;
        y = 5 ;
        angle = 0 ;
        }
…....
}
```

7.5 Exercice

a) Modifier la classe Rectangle en ajoutant le constructeur ci-dessus.
Compiler Rectangle.java. Exécuter MainRectangle. Relever les valeurs affichées. Conclure

b) Ajouter un constructeur qui reçoit comme arguments la longueur et la largeur et qui initialise les coordonnées x et y à 0. Tester le fonctionnement.

c) Ajouter un constructeur qui reçoit comme arguments la longueur, la largeur et les coordonnées x et y. Tester le fonctionnement.

d) Modifier la méthode main de la classe MainRectangle afin de saisir les dimensions du rectangle.

e) Ajouter dans la classe MainRectangle une méthode :
public static Rectangle creerRectangle()
qui
- demande les dimensions et les coordonnées d'un rectangle,
- crée l'objet Rectangle correspondant,
- retourne cet objet Rectangle.

f) Ecrire la méthode agrandir(float a) de la classe Rectangle. Tester son fonctionnement.

g) Ajouter dan la classe Rectangle une méthode
public int compareSurface(Rectangle r)
qui compare la surface du rectangle appelant avec celle du rectangle passé en argument.
Elle retournera :
-1 si la surface du rectangle appelant est inférieure à celle du rectangle passé en argument.
0 si la surface du rectangle appelant est égale à celle du rectangle passé en argument.

+1 si la surface du rectangle appelant est supérieure à celle du rectangle passé en argument.

8. Exercices

Exercice 1 : classe CPoint

Définir et tester une classe CPoint.
Cette classe doit encapsuler correctement 3 attributs :

- les coordonnées du point (abscisse et ordonnée de type réel);
- un compteur qui **compte le nombre de fois où un point est déplacé**.

Elle doit être **sans affichage et sans saisie** (i.e. : les méthodes de la classe CPoint doivent être sans affichage et sans saisie).

Elle doit être munie :

- d'un constructeur avec arguments utilisés pour initialiser les attributs intéressants,
- d'un constructeur sans argument pour initialiser les attributs à 0,
- d'une méthode qui permet de déplacer un point (**l'appel de cette méthode doit constituer le seul moyen pour l'utilisateur de modifier les coordonnées d'un point après sa création**),
- de méthodes permettant d'accéder aux attributs du point (**justifier vos choix**).

Dans le fichier source de la classe CPoint, vous devez **justifier soigneusement** :

- le type et le rôle de chaque attribut;
- le prototype de chaque méthode :
  - type et valeurs possibles de la valeur de retour;
  - type et rôle de chaque paramètre;

Ajouter dans la classe CPoint une fonction membre:
void ajouter(CPoint p)
qui recalcule les coordonnées du point appelant en ajoutant les coordonnées du point donné en paramètre:
x du point appelant = x du point appelant + x du point en paramètre
y du point appelant = y du point appelant + y du point en paramètre

Ajouter dans la classe CPoint une fonction membre:
boolean comparer(CPoint p)
qui retourne true si les coordonnées du point appelant sont égales aux coordonnées du point donné en paramètre.

Exercice 2 : classe CDate

Le calendrier grégorien a été adopté en 1582. Il a permis de corriger la dérive séculaire du calendrier julien alors en usage en ajoutant un jour aux années bissextiles.

Depuis l'ajustement du calendrier grégorien, sont bissextiles les années :
- soit divisibles par 4 mais non divisibles par 100 ;
- soit divisibles par 400.

Donc, inversement, ne sont pas bissextiles les années :
- soit non divisibles par 4 ;
- soit divisibles par 100, mais pas par 400.

Cahier des charges :

Réaliser une classe CDate permettant d'**encapsuler** les caractéristiques (jour, mois, année) d'une date.

La classe CDate devra être **sans affichage et sans saisie** (i.e. : les méthodes de la classe CDate doivent être sans affichage et sans saisie).

Elle devra permettre la gestion de toutes les dates postérieures ou égales au 1/1/1582. Elle devra être capable de **traiter correctement les années bissextiles**.

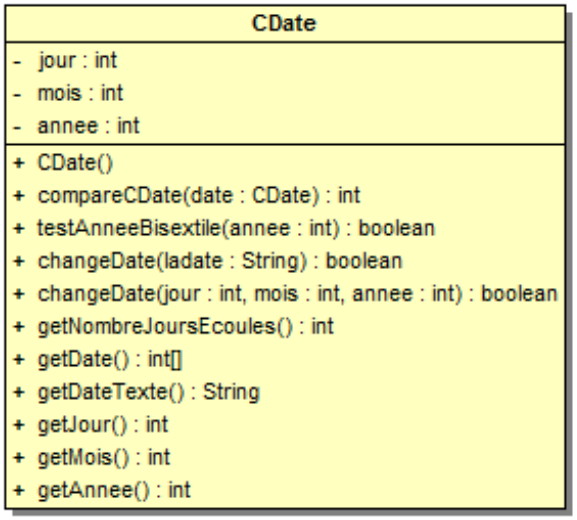
Elle devra disposer :

- d'un constructeur sans paramètre (qui initialise le ou les attribut(s) avec une date valide);
  On peut utiliser la classe LocalDate (java 8) qui permet d'obtenir simplement la date actuelle:

```
LocalDate date = LocalDate.now() ;
int jour = date.getDayOfMonth();
int  mois = date.getMonthValue();
int annee = date.getYear();
```

- de méthodes permettant d'**accéder en consultation** à la date :
  - sous la forme de 3 entiers;
  - ou sous la forme d'une chaîne de caractères aux formats JJ/MM/AAAA, J/M/AAAA, JJ/M/AAAA ou J/MM/AAAA.
- de méthodes permettant d'**accéder en modification** à la date. La nouvelle date pourra être fournie sous la forme de 3 entiers, ou sous la forme d'une chaine de caractères (en respectant les formats précédents). Avant, toute modification du ou des attribut(s), ces méthodes doivent **contrôler la validité de la nouvelle date** : ainsi, le ou les attributs encapsulés dans un objet de cette classe correspondront toujours à une date valide. Il faudra prévoir un mécanisme permettant d'informer l'utilisateur de la validité (ou non validité) de la date transmise en paramètre.
- d'une méthode qui calcule le nombre de jours écoulés depuis le début de l'année "donnée" (l'année de l'objet lui-même).
- d'une méthode qui permet de comparer 2 dates (le paramètre implicite de la méthode et une autre date transmise en paramètre). Quand on compare 2 dates, il y a 3 résultats possibles :
  - d1 antérieure à d2;
  - d1 postérieure à d2;
  - d1 égale à d2.

On propose ici un modèle de la classe CDate :



Réaliser une classe de test avec une méthode *main()* qui teste toutes les méthodes de la classe CDate. Ajouter la méthode **int getNombreJoursEcoules(int annee)** qui retourne le nombre de jours entre le 1er janvier de l'année passée en paramètre et le jour de l'objet lui-même.

Exercice 3 : Le jeu de Chifoumi

Le jeu de Chifoumi s'appelle aussi Caillou-Ciseaux-Papier. Il se joue entre deux joueurs, en général avec les mains. Simultanément, les deux joueurs font un signe avec les mains qui représente soit un caillou, soit des ciseaux, soit un papier.

Nommons A et B les deux joueurs.

- Si les deux joueurs ont fait le même signe, on considère que c'est un cas d'égalité, aucun des deux joueurs ne marque un point.
- Si le joueur A a joué Caillou et le joueur B Ciseaux, A marque un point car "le caillou émousse les ciseaux", et réciproquement.
- Si le joueur A a joué Papier et le joueur B Caillou, A marque un point car "le papier enveloppe le caillou", et réciproquement.
- Si le joueur A a joué Ciseaux et le joueur B Papier, A marque un point car "les ciseaux coupent le papier", et réciproquement.

On demande ici de jouer contre l'ordinateur.

A chaque partie :

- L'utilisateur voit un menu lui permettant de choisir son objet, par exemple taper 0 pour Caillou, 1 pour Papier et 2 pour Ciseaux. Il valide par Entrée.
- Le programme tire alors un nombre au hasard parmi les valeurs 0, 1 et 2, 0 correspondant au Caillou, 1 au Papier et 2 au Ciseaux.
- Le programme doit alors comparer les 2 valeurs et afficher le vainqueur.
- Le vainqueur gagne 1 point par partie.

On impose les entiers 0 pour Caillou, 1 pour Papier et 2 pour Ciseaux.

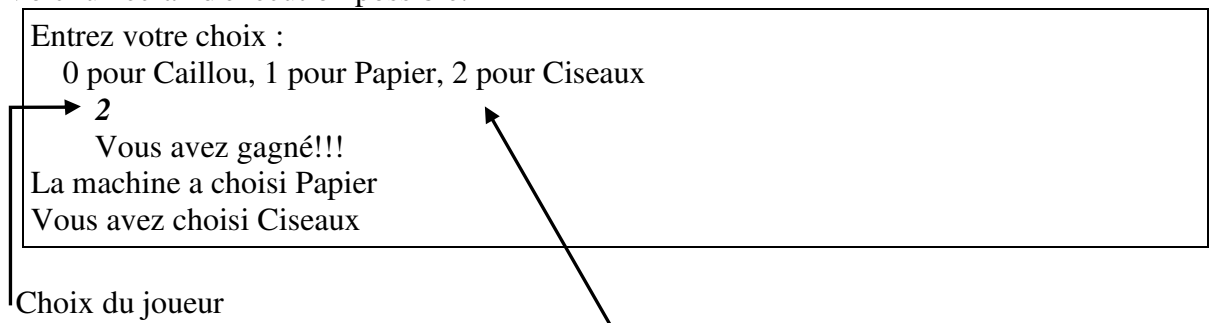
1) Dresser le tableau de vérité du jeu en complétant le tableau suivant.

Joueur	Machine	Résultat pour le joueur
0	0	Egalité
0	1	Perdu
….....	….....	….....

2) On propose ci-dessous le modèle d'une classe Chifoumi en précisant des méthodes et des attributs :

```
class Chifoumi
    int choixMachine;
    int choixJoueur ;
    Scanner clavier;
    public String conversion(int choix)
    public Chifoumi()
    public void afficheChoix()
    public void resultatJeu()
    public void getChoixMachine()
    public void getChoixJoueur()
```

Voici un écran d'exécution possible:



La méthode getChoixJoueur() affiche un message circonstancié puis lit le choix du joueur et le stocke dans choixJoueur.

La méthode getChoixMachine() calcule après un tirage au sort le choix de la machine et le stocke dans choixMachine.

La méthode résultatJeu() calcule le résultat et affiche si le joueur a gagné ou perdu ou s'il y a égalité.

La méthode conversion() retourne un des 3 String suivants "Caillou", "Ciseaux" ou "Papier" en fonction de l'argument -0 ou 1 ou 2- fourni.

La méthode afficheChoix() affiche les messages "La machine a choisi …." et "Vous avez choisi …" en remplaçant les pointillés par les objets réellement choisis.

Le constructeur Chifoumi() crée l'objet clavier pour effectuer la saisie du choix du joueur.

Tirage d'un nombre au hasard avec Java
On utilisera pour cela la méthode statique random() de la classe Math, l'exécution de **Math.random()** retourne un double compris dans l'intervalle [0.0 , 1.0[.
Il faut trouver une solution (simple) permettant d'obtenir un entier dans l'intervalle [0 2] à partir d'un double compris entre 0.0 inclus et 1.0 exclu.
Il est conseillé de visualiser l'aide HTML sur la classe Math de la documentation Java.

Coder la classe Chifoumi.

3) Coder une classe **JouerChifoumi** avec la méthode main utilisée pour créer un objet Chifoumi et exécuter les diverses méthodes de cet objet afin d'effectuer une partie.

4) On donne un nombre de points à atteindre (3 ou 5 par exemple) et le premier joueur qui a atteint ce nombre de points a gagné, chaque partie comptant 1 point.
Modifier la classe **JouerChifoumi** ne contenant que la méthode main() pour jouer par exemple 3 parties.
Il faut récupérer le résultat de chaque partie, une solution consiste à modifier la méthode
void resultatJeu()
par
int resultatJeu()
La méthode retournera par exemple : 0 en cas d'égalité, 1 si la machine a gagné et 2 si c'est le joueur.

5) Modifier la classe **JouerChifoumi** afin de proposer un menu demandant au joueur le nombre de parties qu'il veut faire, s'il veut jouer ou s'il veut arrêter.

9. Les membres (variables ou fonctions) statiques

Dans les exemples précédents, la classe Point ne contient que des membres (méthodes ou attributs) non statiques. Les membres non statiques sont aussi appelés les **membres d'instance**.

Le mot réservé **static** permet de déclarer des **méthodes/attributs de classe**, par opposition aux **méthodes/attributs d'instance**.

Exemple :

```
public class Bicycle{
    private int plate;              // nombre de plateaux
    private int speed;              // nombre de vitesses
    private static int numberOfBicycles = 0;

    public Bicycle(int theplate, int thespeed){
            plate=theplate ;
            speed = thespeed;
            numberOfBicycles++;  // increment number of Bicycles
    }

    // new method to return the plate instance variable
    public int getPlate() {
            return this.plate;
    }

    // accès en modification à l'attribut d'instance plate
    public void setPlate(int plate){
            this.plate = plate;
    }

    // new method to return the numberOfBicycles class variable
    public static int getNumberOfBicycles() {
            return numberOfBicycles;
    }
}
```

Chaque objet de type Bicycle dispose des ses propres attributs plate, et speed, appelés ici **attributs d'instance**.
Tous les objets de type Bicycle "partagent" le même attribut **numberOfBicycles** déclaré **static** appelé **attribut de classe**.

Un membre **static** de classe est accessible par **NomClasse.NomMembreClasse**,
par exemple : Bicycle.numberOfBicycles est accessible sans avoir à instancier d'objet sur la classe.

La méthode getNumberOfBicycles() est appelée **méthode de classe** car elle est déclarée **static**.

Règles d'utilisation :

- Les méthodes d'instance peuvent accéder directement aux attributs d'instance.
- Les méthodes d'instance peuvent accéder directement aux attributs de classe.
- Les méthodes de classe peuvent accéder directement aux attributs de classe.
- Les méthodes de classe ne peuvent accéder directement aux attributs d'instance. Elles doivent passer par un objet instance de la classe, elles ne peuvent pas utiliser le mot réservé this.

Exemple d'utilisation de la classe Bicycle :

```
public class TestBicycle {
        public static void main(String [] args)
        {
                int nb;
                Bicycle a, b;

                // appel de la méthode de classe getNumberOfBicycles()
                nb = Bicycle.getNumberOfBicycles();
                System.out.println("1.Nombre de bicyclettes : " + nb); // Affiche 0 (*1)

                a = new Bicycle(1,1);
                b = new Bicycle(2,2);
                b = new Bicycle(3,3);
                a.setPlate(6);
                System.out.println("Nombre de plateaux de l'objet a : " + a.getPlate());

                // appel de la méthode de classe getNumberOfBicycles()
                nb = Bicycle.getNumberOfBicycles();
                System.out.println("2.Nombre de bicyclettes : " + nb);          // Affiche ... 3
        }
}
```

(*1) : à cet instant de l'exécution du programme, la classe Bicycle n'a pas encore été instanciée ! La seule solution pour utiliser un attribut/methode d'une classe sans avoir a l'instancier est de le/la declarer **static**.

Q11 Tester cet exemple.

10. Les classes "wrapper" (ou enveloppes)

A chaque type primitif (char, byte, short, int , long, float, double, boolean) est associée une classe "wrapper", respectivement :

Character, Byte, Short, Integer, Long, Float, Double, Boolean

Chaque classe "wrapper" encapsule (dans un objet) un attribut du type primitif associé.

De plus, les classes "wrapper" fournissent un ensemble de méthodes permettant de convertir une valeur du type primitif correspondant en String et inversement, ainsi que d'autres méthodes pour manipuler des valeurs du type primitif associé.

Q12 Consulter la documentation de ces classes "wrapper".

Primitive type	Wrapper class
boolean	Boolean
byte	Byte
char	Character
float	Float
int	Integer
long	Long
short	Short
double	Double

Autoboxing : nom d'un mécanisme automatique qui entraine une conversion automatique entre des données de type primitif et les objets de type wrapper correspondants.

Exemple:

```
Integer i ;
i = 5 ;
```

La conversion de l'entier 5 en objet **Integer** est faite automatiquement.

Unboxing : mécanisme automatique inverse à l'autoboxing qui entraine une conversion automatique entre des objets de type wrapper et les données de type primitif correspondants.