

# inf220 Algorithmique

Jean-François Berdjugin, Jean-  
François Remm

IUT1, Département SRC, L'Isle  
d'Abeau

# Référence

- Cours de Jean-François Remm
- <http://www.pps.jussieu.fr/~rifflet/JAVA>

# Ce nous savons

Programmation fonctionnelle :

- Données
- Fonctions

Programme = fonctions +  
données

Pb les données ne sont pas  
corrélées aux fonctions =>  
appliquer les fonctions sur les  
mauvaises données, accéder  
aux fonctions lorsque leur  
nombre augmente, pas de  
nouveaux types, ...

```
Données  
int x  
int y
```

```
Fonctions  
int[] deplacer(int x, int y)  
int[] translater(int x, int y, int dx, int dy)  
..
```

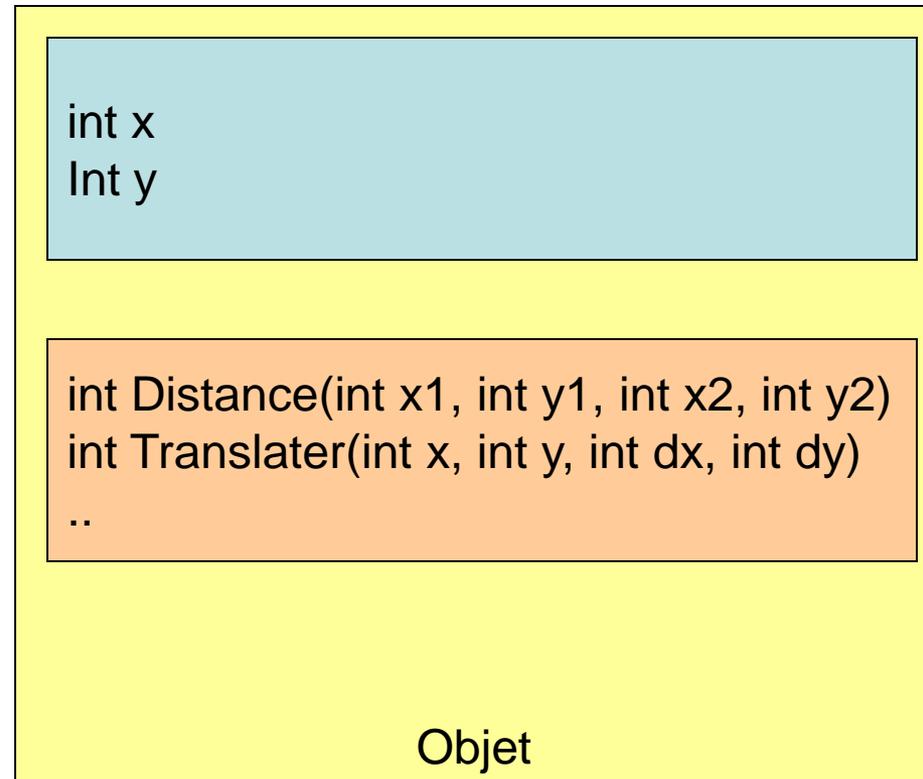
=>

# Programmation orientée objet

Encapsuler les fonctions et les données dans une boîte : l'objet

=>

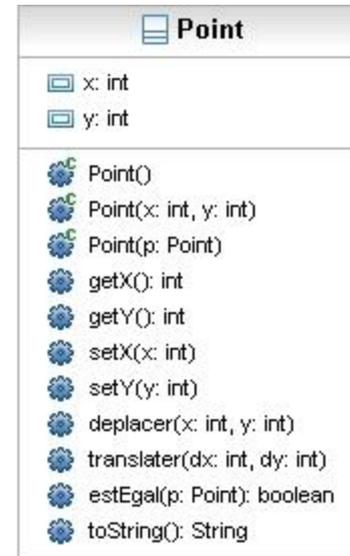
1. Regroupement des données avec leur fonction de traitement
2. Les fonctions (méthodes) modifie l'objet lui même
3. Nouveaux types objets
4. Autre moyen de factoriser le code (Héritage)
5. Nouvelles approches de conception



# Classe

Classe : description  
STATIQUE d'une famille  
d'objets (un type) ayant  
même structure et même  
comportement (le moule  
des objets)

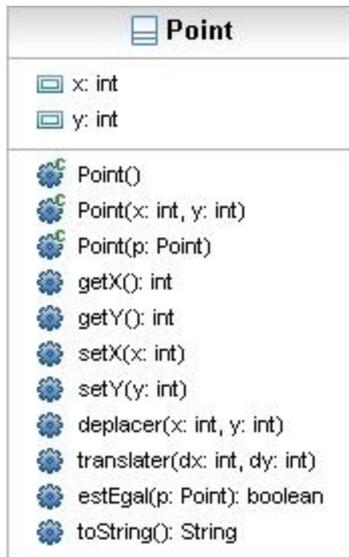
- Objet : entité créée  
DYNAMIQUEMENT, en  
respectant les plans de  
construction donnés par  
sa classe, une instance  
d'une classe.



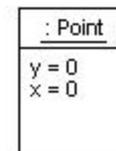
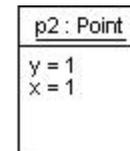
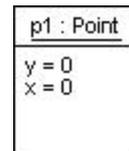
Class Point

# Classe et objets

## Classe

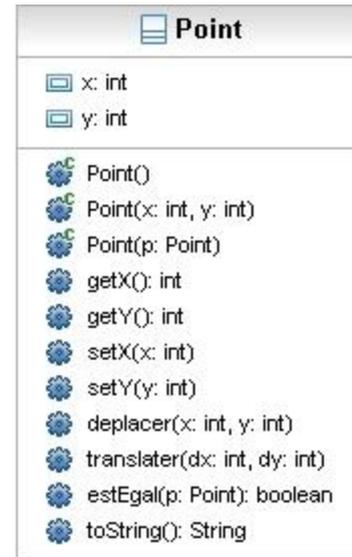


## Objets



# Définir une classe

1. Décrire l'**ETAT** (la structure) d'une instance : variables d'instances (ou propriétés ou attributs) : données caractérisant l'état d'une instance à un moment donné
2. D'écrire son **COMPORTEMENT** :  
Méthodes : fonctions spécialisées dans la manipulation des instances
3. **CONSTRUCTEURS** :  
méthodes spéciales construisant les objets



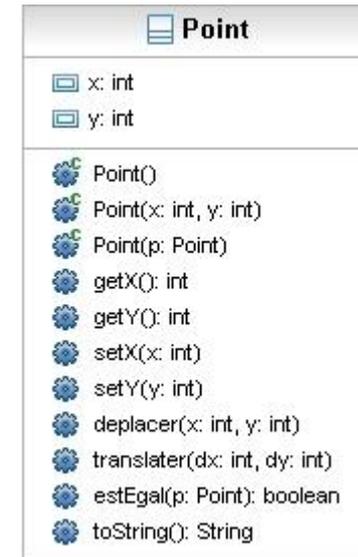
Variables d'instances

Constructeurs

Méthodes

# Définir une classe

1. **ÉTAT** : un point est défini par ces coordonnées x et y
2. **COMPORTEMENT**
  - getX(), getY()
  - setX(int x), setY(int y)
  - déplacer(int x, int y)
  - translater(int dx, int dy)
  - estEgal(Point p)
  - toString()
3. **CONSTRUCTEURS** : le moyen de définir un état initial :
  - Point(), Point(int x, int y), Point(Point p)



# Définir une classe

//code de la class Point

```
public class Point{
```

```
    int x;
```

```
    int y;
```

```
    Point() {
```

```
        x = 0;
```

```
        y = 0;
```

```
    }
```

```
}
```

//code de la class Point

avec restriction de la  
visibilité

```
public class Point{
```

```
    private int x;
```

```
    private int y;
```

```
    Point() {
```

```
        this.x = 0;
```

```
        this.y = 0;
```

```
    }
```

```
}
```

# Accessibilité aux membres d'un objet

accès depuis .... sur un membre	Private	rien	Protected	Public
la classe elle-même	Oui	Oui	Oui	Oui
sous-classe même package	Non	Oui	Oui	Oui
pas une sous-classe, même package	Non	Oui	Oui	Oui
une sous-classe d'un package différent	Non	Non	Oui	Oui
pas une sous-classe d'un package différent	Non	Non	Non	Oui

# Accessibilité aux membres d'un objet

- Pour l'algorithmique toutes nos variables d'instances seront private (-) ☐
- Nous utiliserons des méthodes public (+) pour les manipuler.

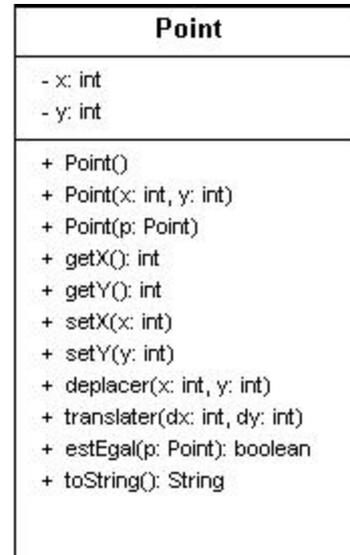


Diagramme de  
classe  
UML1

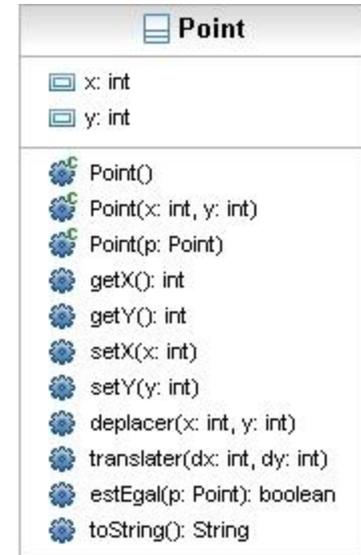


Diagramme de  
classe  
UML2

# Accessibilité aux membres d'un objet

this : permet de désigner l'objet dans lequel on se trouve.

Nous l'utiliserons systématiquement pour accéder aux variables d'instance

```
//code de la class Point plus  
mieux  
public class Point{  
    private int x;  
    private int y;  
    Point() {  
        this.x = 0;  
        this.y = 0;  
    }  
}
```

# Constructeur

Des méthodes particulières de même nom que la classe qui ne retournent rien (même pas void)

Un mécanisme d'instanciation qui initialise les variables d'instance (« les initialisent »).

Sur notre exemple le point est créé en zéro (x=0) zéro (y=0)

```
//code de la class Point
```

```
public class Point{  
    private int x;  
    private int y;  
    Point() {  
        this.x = 0;  
        this.y = 0;  
    }  
}
```

# Surcharge

Permet de définir plusieurs méthodes de même nom différenciées par leurs arguments (nombre, type et position)

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

    public Point() {
        this.x = 0;
        this.y = 0;
    }

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public Point(Point p) {
        this.x = p.x;
        this.y = p.y;
    }
}
```

# surcharge

`void maMethode() //Ok`

`int maMethode() //pas Ok type de retour non pris en compte`

`int maMethode(int a) //Ok`

`int maMethode(int a, int b) //Ok`

`int maMethode(double a, int b) //Ok`

`Int ma Mathode(int a, double b) //Ok`

# Méthodes

Les méthodes fixent le comportement, elles permettent d'obtenir ou de modifier l'état d'un objet.

# Méthodes

Ce n'est qu'une convention mais pour accéder aux variables d'instance, surtout si elles sont privées, l'on définit des « getters » ainsi que des « setters »

```
public int getX() {  
    return this.x;  
}
```

```
public int getY() {  
    return this.y;  
}
```

```
public void setX(int x) {  
    this.x = x;  
}
```

```
public void setY(int y) {  
    this.y = y;  
}
```

# Méthodes

Nous allons réécrire notre en réutilisant au maximum nos propres méthodes.

Un constructeur peut faire appel à un autre constructeur en utilisant **this**(paramètres).

Une méthode peut faire appel à une autre méthode de la classe par son nom. Pour prendre des bonnes habitudes nous utiliserons **this.nom**(paramètres)

# Méthodes

- Nous avons un constructeur « plus généralise » que les autres, nous pouvons le réutiliser

```
public Point(int x, int y) {  
    this.x = x;  
    this.y = y;  
}
```

# Méthodes

```
public Point(int x, int y) {  
    this.x = x;  
    this.y = y;  
}
```

```
public Point() {  
    this(0,0);  
}
```

```
public Point(Point p) {  
    this(p.x,p.y);  
}
```

# Méthodes

Nous pouvons faire de  
même avec la  
translation

```
public void traduire(int  
    dx, int dy)  
{  
    this.x +=dx;  
    this.y +=dy;  
}
```

# Méthodes

```
public void translater(int dx, int dy)
{
    this.x +=dx;
    this.y +=dy;
}
```

//translation sur la diagonale

```
public void translater(int d)
{
    this.translater(d,d);
}
```

# Objets

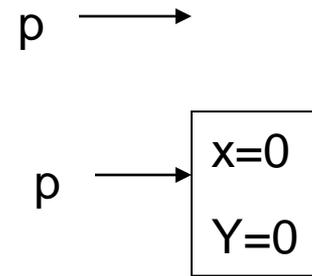
- Les classes ont des instances les objets (ils sont créés dynamiquement).
- Les objet sont créés en utilisant un des constructeurs de la classe
- Le mot clef est **new**

```
public class TestPoint() {  
    public static void main(String[] args) {  
        Point p //p est déclaré comme étant de type point  
        p = new Point(); //p est instancié grâce au constructeur par défaut  
  
        Point p2 = new Point(p); //déclaration est instanciation de p2 au même  
        coordonnées que p  
  
        Point p3 = new Point(10,20); // déclaration de p3 en 10,20  
        Point p4 = new Point(10,20); //déclaration de p4 en 10,20  
    }  
}
```

# Objets

Point p //p est déclaré  
comme étant de type  
point

p = new Point(); //p est  
instancié grâce au  
constructeur sans  
paramètre



# Objets

Point p

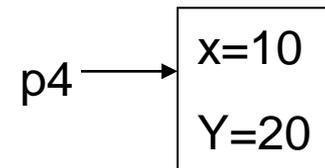
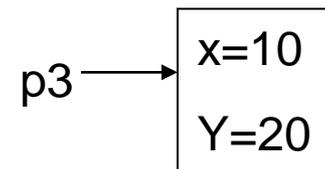
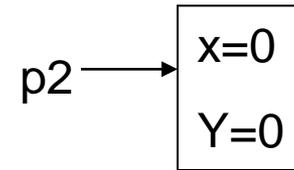
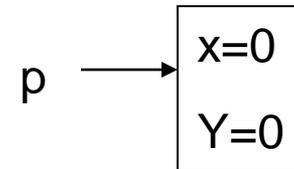
```
p = new Point();
```

```
Point p2 = new Point(p);
```

```
Point p3 = new  
    Point(10,20);
```

```
Point p4 = new  
    Point(10,20);
```

p3=p4 ?



# Objets

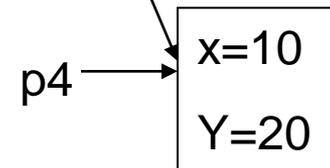
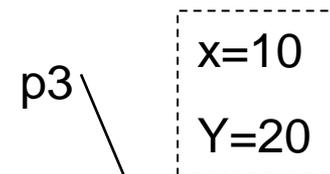
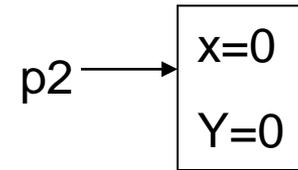
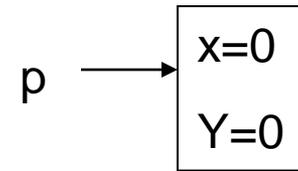
```
Point p  
p = new Point();
```

```
Point p2 = new Point(p);
```

```
Point p3 = new  
    Point(10,20);
```

```
Point p4 = new  
    Point(10,20);
```

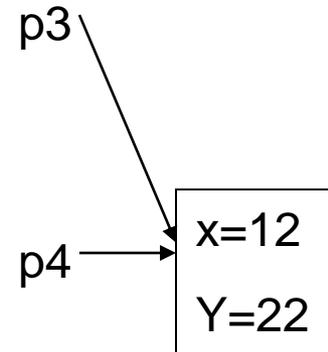
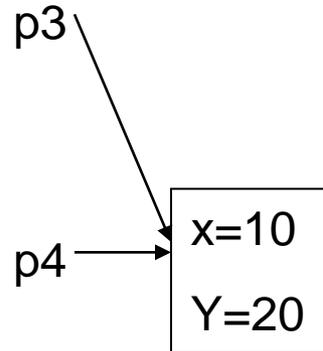
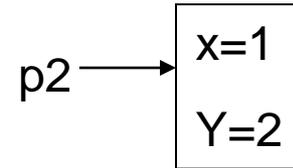
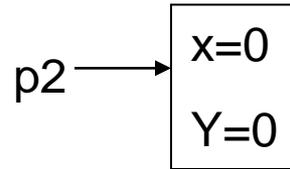
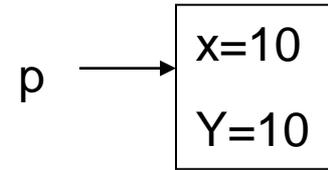
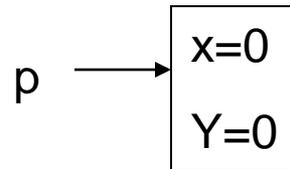
```
p3=p4 !
```



En java  
Garbage  
Collector

# Objets

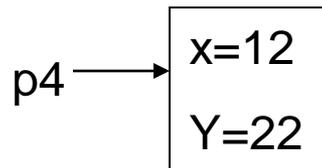
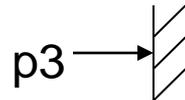
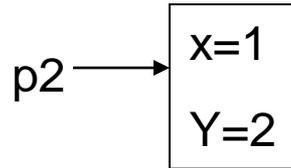
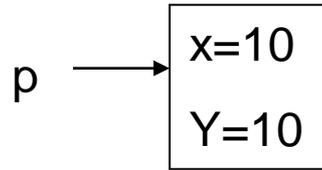
```
p.translater(10);  
p2.translater(1,2);  
p3.translater(1,1);  
p4.translater(1);
```



# Objets

**p3=null**

p3 ne référence  
plus rien



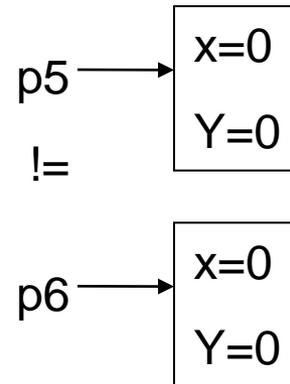
# Objets

```
Point p5 = new Point();
```

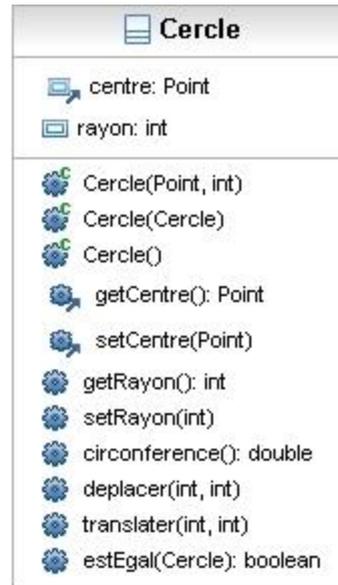
```
Point p6 = new Point();
```

`P6==p5 ?`

Non



# Exercice classe Cercle



# Exercice classe Cercle

```
package cm;

public class Cercle {
    private Point centre;
    private int rayon;

    public Cercle(Point centre, int rayon) {
        this.centre = centre;
        this.rayon = rayon;
    }

    public Cercle(Cercle c)
    {
        this(new Point(c.centre), c.rayon);
    }

    public Cercle()
    {
        this(new Point(),0);
    }

    public Point getCentre() {
        return centre;
    }

    public void setCentre(Point centre) {
        this.centre = centre;
    }

    public int getRayon() {
        return rayon;
    }

    public void setRayon(int rayon) {
        this.rayon = rayon;
    }

    public double circonference()
    {
        return 2*Math.PI*this.rayon;
    }

    public void deplacer(int x, int y)
    {
        this.centre.deplacer(x, y);
    }

    public void translater(int dx, int dy)
    {
        this.centre.translater(dx, dy);
    }

    public boolean estEgal(Cercle c)
    {
        return this.rayon == c.rayon && this.centre.estEgal(c.centre);
    }
}
```

# Exercice classe Quadrilatère



# Exercice classe Quadrilatère

```
package cm;

public class Quadrilataire {
    private Point[] points;

    public Quadrilataire(Point[] points) {
        this.points = new Point[points.length];
        for (int i = 0; i < points.length; i++)
        {
            this.points[i] = new Point(points[i]);
        }
    }

    public Quadrilataire(Point p1, Point p2, Point p3, Point p4)
    {
        this.points = new Point[4];
        this.points[0] = new Point(p1);
        this.points[1] = new Point(p2);
        this.points[2] = new Point(p3);
        this.points[3] = new Point(p4);
    }

    public void translater(int dx, int dy)
    {
        for (int i = 0; i < this.points.length; i++)
            this.points[i].translater(dx, dy);
    }
}
```

# Reste à voir

- Lors du cours de programmation nous compléterons par :
  - L'héritage : un moyen de factoriser du code
  - Le polymorphisme : un changement de type à l'exécution
  - Les variable et méthodes de classe
  - La généricité : la possibilité de passer un type en paramètre
  - ...
  - Swing