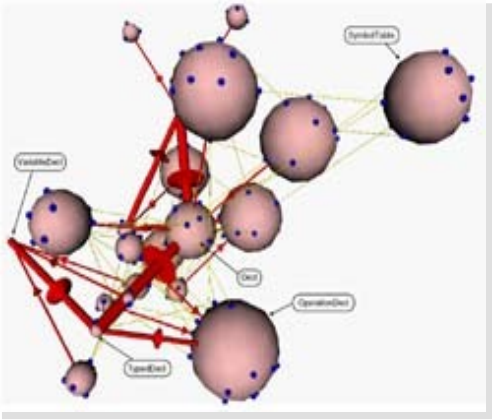
La programmation Orientée Objets et le langage Java

Belgacem BEN HEDIA
ESISAR-INP Grenoble

CITI - INSA Lyon
EPT



Cours 2: « L'encapsulation »

belgacem.ben-hedia@insa-lyon.fr
belgacem.ben-hedia@esisar.inpg.fr
http://www.bbhedia.org

La programmation orientée objets

- Concepts apparus à partir des années 60
- premiers langages : Simula (simulation), Smalltalk (IA, IHM, etc.)
- convergence de deux courants de recherche
  - courant *génie logiciel* :

  lutter contre la complexité des gros programmes

  problématique : *correction, robustesse, réutilisabilite,* etc.
  - courant *représentation des connaissances* :

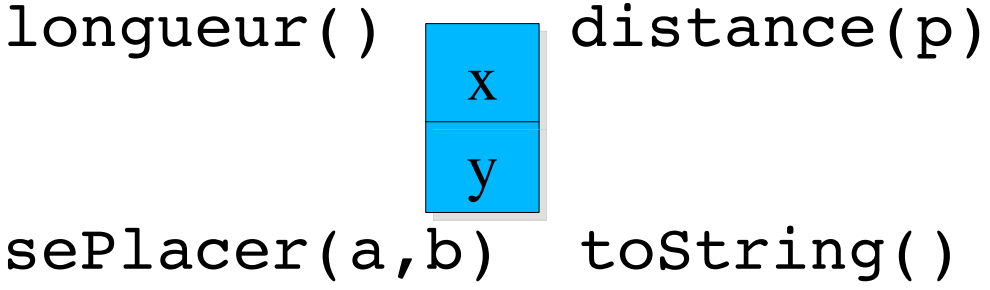
  problématique : représenter les concepts (*cette chose est une Table*), l'abstraction (*toute Table est une sorte de Meuble*)
- principales notions
  - encapsulation
  - héritage
  - polymorphisme, redéfinition des méthodes, exceptions, généricité, etc.

Encapsulation: objets et messages

Esprit de la chose : objets

- un programme est un ensemble *d'objets*
- chaque objet possède un état (défini par un ensemble de valeurs changeantes) et un comportement (ensemble de *méthodes*)
- le comportement *encapsule* (*entoure, cache, protège*) l'état

Exemple, des objets Point pour représenter les points du plan :

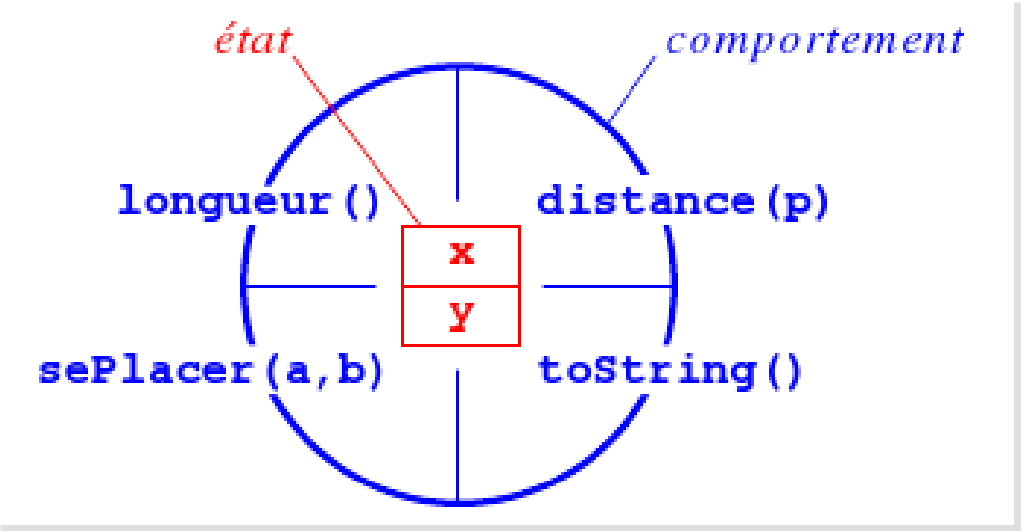


Encapsulation: objets et messages

Esprit de la chose : objets

- un programme est un ensemble *d'objets*
- chaque objet possède un état (défini par un ensemble de valeurs changeantes) et un comportement (ensemble de *méthodes*)
- le comportement *encapsule* (*entoure, cache, protège*) l'état

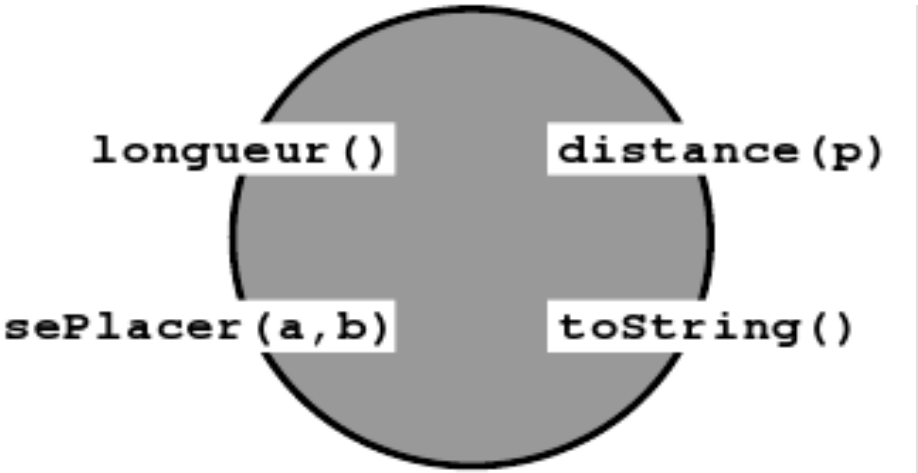
Exemple, des objets Point pour représenter les points du plan :



Encapsulation: objets et messages

Esprit de la chose : messages

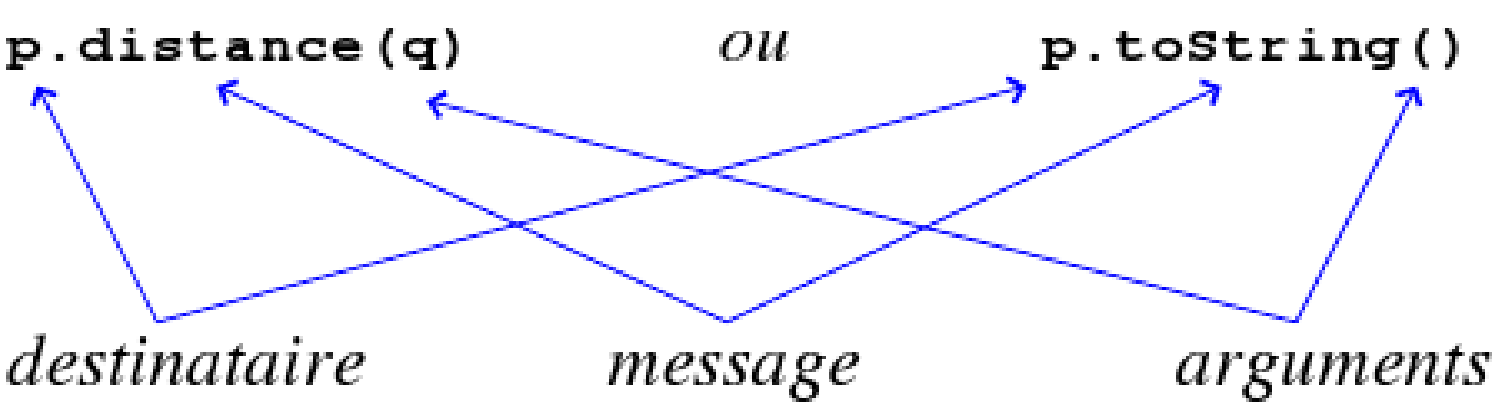
- les objets sont *opaques* et interagissent par *envoi* de messages
- un message est « personnalisé » : il a toujours un *destinataire* (et parfois des *arguments*)
- un message est « abstrait » : il dit *quoi* faire, mais non *comment* (le destinataire connaît les détails précis de ce qu'on lui demande)
- *interface* d'un objet : ensemble des messages qu'il « comprend », c'est l'aspect public de l'objet



Objets et messages en Java

Comment cela se présente en Java

- objets : des structures contenant des variables et du code
- état : des variables, appelées *champs* ou *variables d'instance*
- comportement : des fonctions, appelées *méthodes d'instance*
- envoi de message : appel de méthode, sous la forme



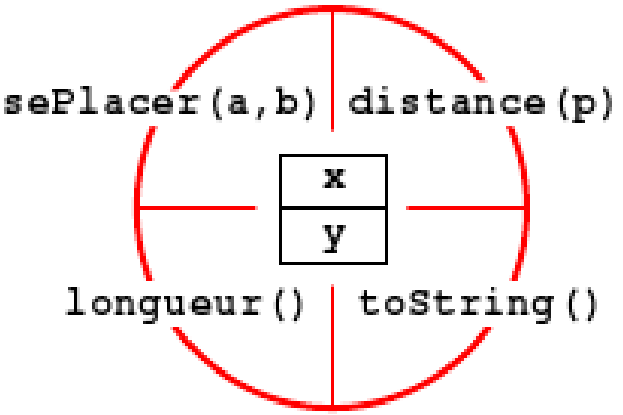
au lieu de : `distance(p,q)` ou `tpString(p)`

- Exemple : une classe Point pour représenter les points du plan

La classe Point

Fichier Point.java

```
class Point {
    int x, y ;
    void sePlacer(int a, int b) {
        « validation de a et b »
        x = a ;
        y = b ;
    }
    double longueur() {
        return Math.sqrt(x * x + y * y) ;
    }
    double distance(Point p) {
        int dx = x - p.x ;
        int dy = y - p.y ;
        return Math.sqrt(dx * dx + dy * dy) ;
    }
    public String toString() {
        return "(" + x + ", " + y + ")" ;
    }
    ...
}
```

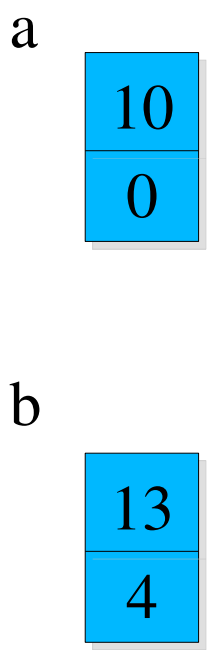


- Noter que le qualifieur `static` a disparu.

Utilisation de la classe Point

Fichier TestPoint.java

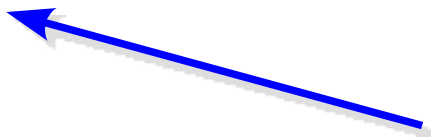
```
class TestPoint {
    public static void main(String[] args) {
        Point a, b ;

        a = new Point() ;
        b = new Point() ;

        a.sePlacer(10, 0) ;
        b.sePlacer(13, 4) ;

        System.out.println(a.longueur()) ;
        System.out.println(a.distance(b)) ;
        System.out.println(a.toString()) ;
        System.out.println(b) ;
    }
}
```



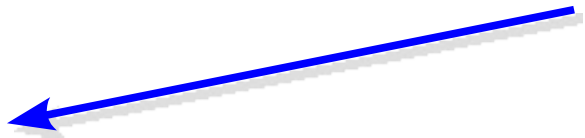

```
Execution :
        10.0
        5.0
        (10, 0)
        (13, 4)
```

Faut il dire *objet* ou bien *classe* ?

```
class Point {
   int x, y;
   void sePlacer(int a, int
b) {...}
   double longueur() {...}
   ...
}
...
Point a = new Point();
Point b = new Point();
...
```



La classe



des objets

- classe : aspect descriptif, statique, unique
- objet : aspect existentiel, dynamique, multiple
- le bien parler :
  - [la valeur de] a est un [objet] Point
  - [la valeur de] a est une instance de la classe Point
  - `new` est l'opération d'instanciation (= création d'un objet)

Accès aux Membres d'instance, notation

- Pour accéder à un membre d'instance on doit mentionner l'instance en question :

  « `unObjet.unMembre` »

- Dans le cas d'une méthode : version Java de l'envoi de message :

  `a.longueur()` envoi du message longueur à l'objet `a`

- à l'intérieur d'une telle méthode, l'objet est implicite :

```
class Point {
      int x, y ;
      ...
      double longueur() {
         return Math.sqrt(x * x + y * y) ;
      }
}
```

- qui sont ces x et y qui apparaissent dans la méthode d'instance précédente ?

réponse : les variables de l'objet à travers lequel on aura appelé la méthode

```
Point a = new Point() ;
...
double r = a.longueur() ;
```

et alors le corps de la fonction équivaut à `return Math.sqrt(a.x * a.x + a.y * a.y) ;`

Membres d'instance et membres de classe

- membres : les *variables* (ou champs) et les méthodes
- les *membres d'instance* sont attaches aux instances :
  - variable d'instance : chaque objet possède son propre exemplaire

  (créer un objet c'est allouer ces variables ; `new` fait cela)
  - méthode d'instance : on doit l'appeler à travers un objet

  (le destinataire de l'envoi de message) : `a.longueur()`
- les *membres de classe* ne sont pas attaches aux instances :

  ils n'appartiennent qu'à la classe :
  - ils sont signales par le qualifieur `static`
  - *variable de classe* : il y en a un exemplaire unique, accessible par tous les objets de la classe
  - *méthode de classe* : on peut l'appeler seule, comme une fonction C (mais il faudra préciser la classe)

Membres d'instance et membres de classe

Exercice de style

Fichier Point.java

```
class Point {
        int x, y ;
        ...
        double distance(Point p) { // méthode d'instance
            int dx = x - p.x ;
            int dy = y - p.y ;
            return Math.sqrt(dx * dx + dy * dy) ;
        }
        static double distance(Point u, Point v) { // méthode de classe
            int dx = u.x - v.x ;
            int dy = u.y - v.y ;
            return Math.sqrt(dx * dx + dy * dy) ;
        }
}
```

Emploi(dans une autre classe) :

```
Point a, b ;
...
r = a.distance(b) ;          // appel de la méthode d'instance
r = Point.distance(a, b) ;   // appel de la méthode de classe
```

Accès à ses propres membres, accès « à soi-même »

- dans une méthode d'instance, l'*objet à travers lequel on appelle la méthode* est désigné implicitement :

```
double distance(Point p) {
   int dx = x - p.x ;
   int dy = y - p.y ;
   return Math.sqrt(dx * dx + dy * dy) ;
}
```

- cet objet peut aussietre designe explicitement : `this`

```
double distance(Point p) {
   int dx = this.x - p.x ;
   int dy = this.y - p.y ;
   return Math.sqrt(dx * dx + dy * dy) ;
}
```

dans un appel tel que `a.distance(b)` on aura this ≡ a (et p ≡ b)

Accès à ses propres membres, accès « à soi-même »

- *Exercice de style 1*. Écrire la méthode distance de classe en utilisant la méthode d'instance :

```
double distance(Point p) {
    int dx = x - p.x ;
    int dy = y - p.y ;
    return Math.sqrt(dx * dx + dy * dy) ;
}
static double distance(Point p, Point q) {
    return p.distance(q) ;
}
```

- *Exercice de style 2*. Écrire la méthode distance d'instance en utilisant la méthode de classe :

```
static double distance(Point p, Point q) {
    int dx = p.x - q.x ;
    int dy = p.y - q.y ;
    return Math.sqrt(dx * dx + dy * dy) ;
}
double distance(Point p) {
    return distance(this, p) ;
}
```

Constructeurs

- Constructeurs : méthodes
  - ayant même nom que la classe
  - n'ayant pas pas de type de retour ni d'instruction return
- leur rôle : initialiser l'espace qui vient d'être alloue pour l'objet

```
class Point {
   int x, y ;
   Point(int a, int b) {
      « validation de a et b »
      x = a ;
      y = b ;
   }
   ...
}
```

- emploi (`p` étant une variable de type `Point`) :

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

- s'emploie souvent associe à une déclaration :

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

Surcharge des constructeurs

Fichier Point.java

```
class Point {
    int x, y ;
    Point(int a, int b) {
        « validation de a et b »
        x = a ;
        y = b ;
    }
    Point(int a) {
        « validation de a »
        x = a ;
        y = 0 ;
    }
    Point() {
        x = 0 ;
        y = 0 ;
    }
    ...
}
```

Emplois respectifs :

```
Point p = new Point(u, v) ;
...
Point q = new Point(w) ;
...
Point r = new Point() ;
```

A retenir : la création d'un objet se fait toujours

- par l'opérateur `new`
- associe à l'appel d'un constructeur

Ramener un constructeur à un autre

Fichier Point.java

```
class Point {
    int x, y ;
    Point(int a, int b) {
        « validation de a et b »
        x = a ;
        y = b ;
    }
    Point(int a) {
        « validation de a »
        this(a,0) ;

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

    }
    ...
}
```

Emplois respectifs :

```
Point p = new Point(u, v) ;
...
Point q = new Point(w) ;
...
Point r = new Point() ;
```

Attention : « `this(arguments);` » doit être *la première instruction d'un constructeur*

Le constructeur *offert par la maison*

- « la construction d'un objet implique toujours l'appel d'un constructeur »

pourtant notre première classe fonctionnait. Comment ?

si le programmeur n'a écrit aucun constructeur alors le compilateur fournit un constructeur

  - sans arguments
  - entièrement fait d'initialisations par défaut

- par exemple

la classe

```
public class Point {
      int x, y ;
}
```

équivaut à la classe

```
public class Point {
      int x, y ;
      Point() {
         x=y=0 ;
      }
}
```

- ce « cadeau » cesse dès qu'un constructeur quelconque est explicite

Initialisation des variables d'instance : récapitulation

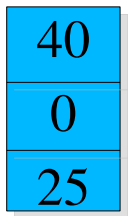
Fichier Recipient.java

```
class Recipient {
      float largeur = 40 ;
      float hauteur ;
      float niveau ;
      ...
      Recipient(float niveau) {
         this.niveau = niveau ;
      }
      ...
}
```

Emploi :

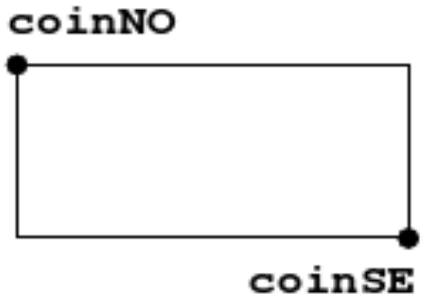
```
Recipient rec = new Recipient(25) ;
```

rec



Objets contenant d'autres objets

- Les objets peuvent avoir (ont souvent) pour membres d'autres objets
- Exemple : un objet `Rectangle` est défini par deux objets `Point`



Bonne nouvelle : il n'y a pas de problème particulier pour l'initialisation :

Fichier Rectangle.java

```
class Rectangle {
        Point coinNO, coinSE ;
        Rectangle(int a, int b, int c, int d) {
                coinNO = new Point(a, b) ;
                coinSE = new Point(c, d) ;
        }
        ...
}
```

A propos, attention...

Mise en garde sur une erreur grave que le compilateur ne peut pas déceler :

Fichier Rectangle.java

```
class Rectangle {
       Point coinNO, coinSE ;
       Rectangle(int a, int b, int c, int d) {
       Point coinNO = new Point(a, b) ;
       Point coinSE = new Point(c, d) ;
}
...
public static void main(String[] args) {
   Rectangle r = new Rectangle(1, 2, 3, 4) ;
   System.out.println(r.coinNO + " :" + r.coinSE) ;
}
}
```

On obtient invariablement (0,0):(0,0)

Initialisation des variables de classe

- crées lors de la création des objets (`new`), les variables d'instance peuvent être initialisées dans les constructeurs
- les variables de classe sont crées au lancement du programme ; comment les initialiser ?
- autant que possible, utiliser des affectations

```
public class Recipient {
    static int n = 64 ;
    static String[] jours = { "lundi", "mardi", ...
    "dimanche" } ;
...
```

- mais comment obtenir ceci ?

```
static long[] puis2 =
{ 1, 2, 4, 8, 16, 32, 64, 128, ... 18446744073709551616} ;
```

(il y a n valeurs, n défini plus haut)

Initialisation des variables de classe

- « bloc statique » : morceau de code `a executer lors du chargement de la classe

```
public class Recipient {
    static int n = 64 ;
    static String[] jours = { "lundi", "mardi", ...
    "dimanche" } ;
    ...
    static long[] puis2 = new long[n]
    static {
        puis2[0] = 1 ;
        for (int i = 1 ; i < n ; i++)
        puis2[i] = 2 * puis2[i - 1] ;
    }
    ...
```

Droits d'accès au membres de classes

- Question : quels membres d'un objet a-t-on le *droit* de mentionner ?
- il s'agit d'un droit *relatif* : la question se pose lors de l'écriture d'une méthode, il faut savoir dans quelle classe se trouve cette dernière
- un membre (variable ou méthode, d'instance ou de classe) peut être qualifié

	ils est alors accessible
`private`	depuis les méthodes de la même classe...
(rien)	...et les méthodes des classes du même paquet...
`protected`	...et les méthodes des sous-classes (cf. cours 3)...
`public`	...et les méthodes de toutes les classes

- cela n'a rien à voir
  - avec la notation : le membre x de l'objet p s'écrit `p.x` qu'on ait ou non le droit d'y accéder
  - avec la directive `import` (qui ne concerne que la manière de nommer les classes)

Droits d'accès, exemple

Fichier Point.java

```
class Point {
    private int x, y ;
    public void placer(int x, int y) {
        « validation de x et y »
        this.x = x ;
        this.y = y ;
    }
    public Point(int x, int y) {
        placer(x, y) ;
    }
    public int getX() {
        return x ;
    }
    public int getY() {
        return y ;
    }
    ...
}
```

Bénéfice retiré

- un point ne peut être construit que par `new Point(a,b)`
- il ne peut être modifié que par `p.placer(a,b)`
- donc on peut garantir que tout point est constamment valide

A propos d'accesseurs

En quoi la configuration

Fichier Point.java

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

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

est elle meilleure que

Fichier Point.java

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


    ...
}
```

- la consultation de x et y depuis l'extérieur de la classe `Point` est rendue possible dans les deux cas
- dans le premier cas la modification de x ou y est impossible

La vue publique d'une classe

- Ce que les utilisateurs ont besoin de savoir (ailleurs on dit « interface »)

L' *interface* (au sens de *face publique* de la classe `Point`

```
class Point {
            /* création d'un point de coordonnées x et y */
        public Point(int x, int y) ;
            /* modification des coordonnées d'un point */
        public void placer(int x, int y) ;
            /* abscisse du point */
        public int getX() ;
            /* ordonnée du point */
        public int getY() ;
        ...
}
```

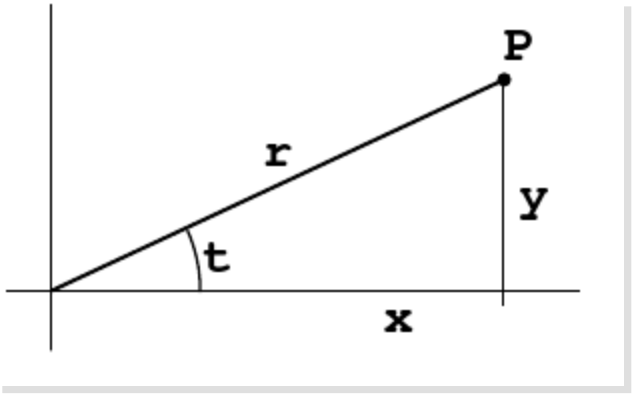
- *Encapsuler :* cacher les détails internes, pour
  - garantir la correction des objets
  - garantir qu'aucune application ne s'appuie sur ces détails
  - qui peuvent donc être changes sans provoquer d'erreur

Encapsulation

- L'encapsulation produit l'indépendance de l'implémentation Cela permet de changer le dedans (l'implémentation) des classes

Fichier Point.java

```
class Point {
    private int r, t ;
    public void placer(int x, int y) {
        r = Math.sqrt(x * x + y * y) ;
        t = Math.atan2(y, x) ;
    }
    public Point(int x, int y) {
        placer(x, y) ;
    }
    public int getX() {
        return r * Math.cos(t) ;
    }
    public int getY() {
        return r * Math.sin(t) ;
    }
    ...
}
```



- Aucune application utilisant la précédente version de `Point` n'est devenue obsolète

Classes publiques et « par défaut »

- les classes avec *visibilité par défaut* (i.e. non qualifié `public`) sont accessibles uniquement depuis les classes de leur propre paquetage

```
package mesAffaires.outils ;
class Machin {
    ...
}
```

- les classes qualifié public sont accessibles depuis les classes de tous les paquetages

```
package mesAffaires.outils ;
public class Truc {
    ...
}
```

- les classes `public` imposent le chemin « chemin » du fichier qui les contient :
  - la classe `Truc` doit se trouver dans un fichier nomme `Truc.java`
  - place dans le dossier `./mesAffaires/outils`

Variables finales

Une variable (d'instance ou de classe) qualifié final ne doit pas changer de valeur

- variable d'instance : reçoit une valeur à la construction et n'en change plus
- variable de classe : se comporte comme les constantes des autres langages

Fichier Point.java

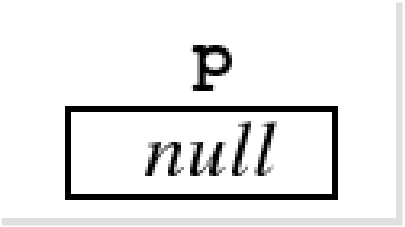
```
public class Point {
        private                 int x, y ;
        private static          int nombreDePoints = 0 ;
        private           final int rang ;
        private static    final int MAX POINTS = 1000 ;

        public Point(int x, int y) {
                if (nombreDePoints > MAX POINTS)
                    signaler une erreur « trop de points »
                this.x = x ;
                this.y = y ;
                rang = nombreDePoints ;
                nombreDePoints++ ;
        }
        ...
}
```

Accès par *références* aux objets

Les objets sont *toujours* désignes par référence

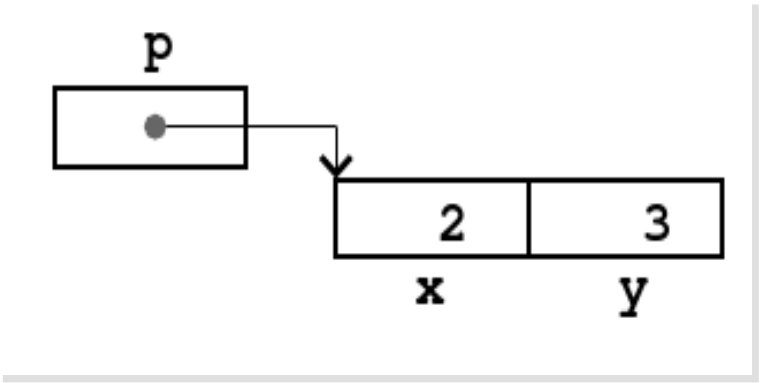
- la déclaration d'une variable de type objet ne fournit rien de plus qu'un pointeur *null* :

```
Point p ;
```



- l'objet n'existe qu'à partir de l'opération `new` qui le crée :

```
p = new Point(2, 3) ;
```

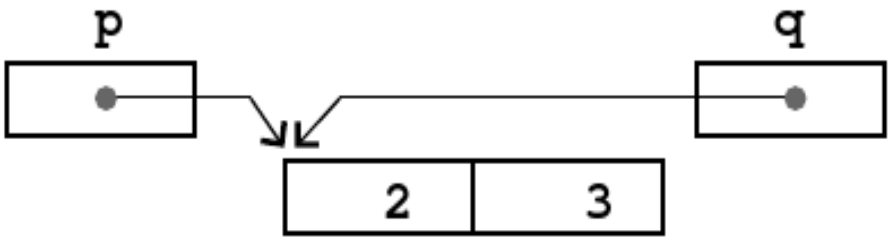


Accès par *références* aux objets

- Conséquence sur la « duplication » des objets

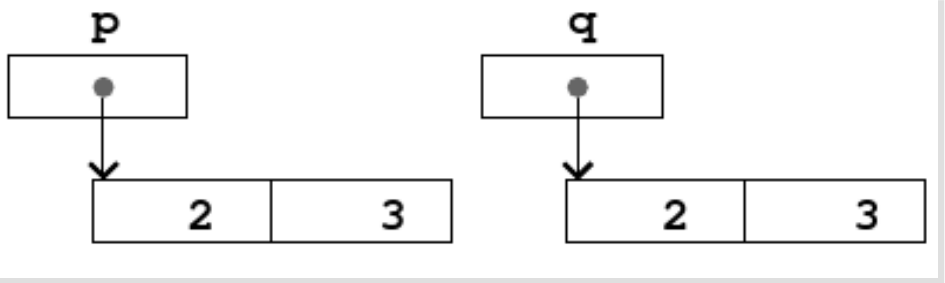
```
Point p = new Point(2, 3) ;
...
Point q = p ;
```

- par la suite, `p` et `q` semblent liés. La raison :



- pour dupliquer effectivement un objet :

```
q = (Point) p.clone() ;
```

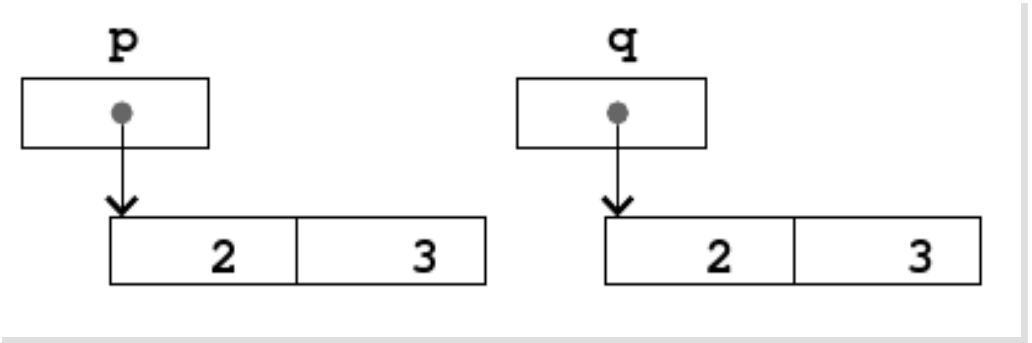


Accès par *références* aux objets

- Conséquence sur la comparaison des objets

```
Point p = new Point(2, 3) ;
Point q = new Point(2, 3) ;
...
if ( p == q )
   System.out.println("egaux") ;
...
```

- le test se révèle faux. La raison :



- la bonne comparaison

```
if ( p.equals(q) )
   System.out.println("egaux") ;
...
```

A propos d'« orienté objets »

- Programmation traditionnelle (ou « orientée actions ») :
  - début de la réflexion, les actions : « qu'est-ce qu'on va faire »
  - plus tard apparaissent les données : « sur quoi ces actions porteront ? »
  - *les objets sont à l'intérieur des actions*

    (ils en sont les paramètres, les résultats, etc.)
- La programmation orientée objets inverse les termes :
  - le début de la réflexion concerne les objets
  - les actions apparaissent comme attributs (le comportement) des objets
  - les actions sont à *l'intérieur* des objets
- très adaptée aux problèmes avec de nombreuses structures de données (exemple : les interfaces graphiques)
- tous les problème ne tirent pas forcement profit de la POO

(exemple : déguiser de purs algorithmes en objets est un peu futile)