Classes et Objets
(1ère partie)

dernière mise à jour 12/12/2024 14:38

Philippe Genoud
Philippe.Genoud@imag.fr

Plan

- Classes
- Objets
- Références
- Création d'objets
  - *constructeur par défaut*
  - *gestion mémoire*
- Accès aux attributs d'un objet
- Envoi de messages
- this : l'objet "courant"
- Objets et encapsulation

Classe

- Une classe est constituée de descriptions de :
  - *données : que l'on nomme **attributs**.*
  - *traitements (procédures et fonctions) : que l'on nomme **méthodes***

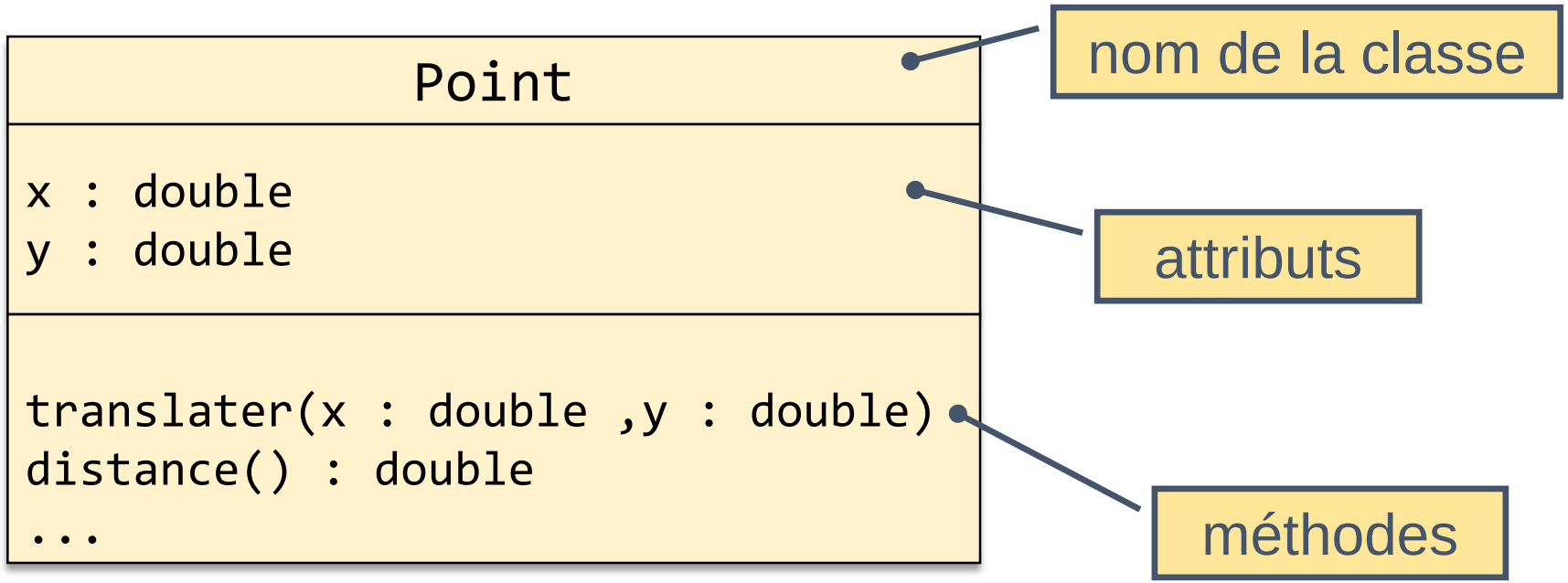
- Une **classe** est un **modèle** de définition pour des objets ayant une sémantique commune.
  - *ayant même structure (même ensemble d'attributs),*
  - *ayant même comportement (mêmes opérations, méthodes),*

- Les **objets** sont des représentations **dynamiques** (instanciation), du modèle défini pour eux au travers de la classe.
  - *Une classe permet d'**instancier** (créer)plusieurs objets*
  - *Chaque objet est **instance** d'une (seule) classe*

Classe

Notation UML

http://uml.free.fr



Classe

fichier **Point.java**

Le nom doit être identique au nom de la classe

Syntaxe JAVA

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

   void translater(double dx,double dy) {
       x += dx;
       y += dy;
   }

   double distance() {
       double dist;
       dist = Math.sqrt(x * x + y * y);
       return dist;
   }

}
```

nom de la classe

Attributs
(fields)

Membres de la classe

Méthodes

Syntaxe JAVA : visibilité des attributs

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

    void translater(double dx, double dy){
        x += dx;
        y += dy;
    }

    double distance() {
        double dist;
        dist = Math.sqrt(x * x + y * y);
        return dist;
    }
}
```

Les attributs sont des variables « globales » à l'unité syntaxique que constitue la classe :
ils sont accessibles dans toutes les méthodes de la classe.

Classe

Une classe Java constitue un **espace de nommage**

Deux classes différentes peuvent avoir des membres de nom identique

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

   void translater(double dx, double dy){
         x += dx;
         y += dy;
   }

   double distance() {
         double dist;
         dist = Math.sqrt(x*x+y*y);
         return dist;
   }
}
```

```
class Cercle {
   double x;  // abscisse du centre
   double y;  // ordonnée du centre
   double r;  // rayon

   void translater(double dx, double dy){
         x += dx;
         y += dy;
   }

   ...
}
```

Les membres d'une classe seront accédés via des objets. Selon le type de l'objet (Point ou Cercle), Java saura distinguer à quels attributs ou méthodes il est fait référence

Déclaration des attributs

Une déclaration d'attribut est de la forme :

```
type nomAttribut;
```

ou bien

```
type nomAttribut = expressionInitialisation;
```

type simple (pas Objet) :
char
int
byte
short
long
double
float
boolean

type structuré (Objet) :
type est le nom d'une classe connue dans le contexte de compilation et d'exécution

Nécessaire si votre classe utilise une autre classe d'un autre package et qui n'est pas le package **java.lang**

```
import java.awt.Color;

class Point {
   double x = 0;
   double y = 0;
   Color c;

   ...
}
```

Un point a une couleur définie par un objet de type **Color**

Déclaration des méthodes

- *« Une déclaration de méthode définit du code exécutable qui peut être invoqué, en passant éventuellement un nombre fixé de valeurs comme arguments »* The Java Langage Specification J. Gosling, B Joy, G. Steel, G. Bracha

- Déclaration d'une méthode

```
<typeRetour> nomMethode( <liste de paramètres> ) {
    <corps de la méthode>
}
```

- Exemple

Signature de la méthode

```
double min(double a, double b) {
    if (a < b)
       return a;
    else
       return b;
}
```

⇔

```
double min(double a, double b) {
   return (a<b)?a:b ;
}
```

Expression conditionnelle

Déclaration des méthodes

```
<typeRetour> nomMethode( <liste de paramètres> ) {
    <corps de la méthode>
}
```

- **<typeRetour>**
  - Quand la méthode renvoie une valeur (fonction) indique le type de la valeur renvoyée (type simple ou nom d'une classe)

```
double min(double a, double b)    // renvoie un nombre flottant double précision
int[] premiers(int n)             // renvoie un tableau d'entiers
Color getColor()                  // renvoie une référence d'objet Color
```

  - **void** si la méthode ne renvoie pas de valeur (procédure)

```
void afficher(double[][] m)
```

Déclaration des méthodes

```
<typeRetour> nomMethode( <liste de paramètres> ) {
    <corps de la méthode>
}
```

- **<liste de paramètres>**
  - vide si la méthode n'a pas de paramètres

```
int lireEntier()
void afficher()
```

  - une suite de couples *type identificateur* séparés par des virgules

```
double min(double a, double b)
int min(int[] tab)
void setColor(Color c)
```

```
<typeRetour> nomMethode( <liste de paramètres> ) {
    <corps de la méthode>
}
```

- **<corps de la méthode>**
  - suite de déclarations de variables locales et d'instructions
  - si la méthode à un type de retour le corps de la méthode doit contenir **au moins** une instruction **`return`** ***`expression`*** où ***`expression`*** délivre une valeur compatible avec le type de retour déclaré.

Les types doivent correspondre

```
double min(double a, double b) {
    double vMin;          ← Variable locale
    if (a < b)
       vMin = a;
    else
       vMin = b;
    return vMin;          ← Instruction de retour
}
```

Déclaration des méthodes

- Si la méthode à un type de retour le corps de la méthode doit contenir au moins une instruction **`return expression`**

```
boolean contient(int[] tab, int val) {
    boolean trouve = false;
    int i = 0;
    while ((i < tab.length) && (! trouve)) {
        if (tab[i] == val){
            trouve = true;
        }
        i++;
    }
    return trouve;
}
```

```
for (int i = 0; i < tab.length; i++){
    if (tab[i] == val)
        return true;
}
return false;
```

- Possibilité d'avoir plusieurs instructions **`return`**
- Lorsqu'une instruction **`return`** est exécutée retour au programme appelant
  - Les instructions suivant le **`return`** dans le corps de la méthode ne sont pas exécutées

- `return` sert aussi à sortir d'une méthode sans renvoyer de valeur (méthode ayant `void` comme type retour)

```
void afficherPosition(int[] tab, int val) {
    boolean trouve = false;
    int i = 0;
    while ((i < tab.length) && (tab[i] != val)) {
       i++;
    }
    if (i == tab.length) {
       System.out.println(val + " n'est pas présente dans le tableau");
    }
    else {
       System.out.println("La position de " + val + " est " + i);
    }
}
```

```
for (int i = 0; i < tab.length; i++) {
   if (tab[i] == val){
     System.out.println("La position de " + val + " est " + i);
     return;
   }
}
System.out.println(val + " n'est pas présente dans le tableau");
```

- *<corps de la méthode>*
  - suite de déclarations de **variables locales** et d'instructions

- Les variables locales sont des variables déclarées à l'intérieur d'une méthode
  - conservent les données qui sont manipulées par la méthode
  - ne sont accessibles que dans le bloc dans lequel elles ont été déclarées
  - leur valeur est perdue lorsque la méthode termine son exécution

```
void methode1(...) {
    int i;
    double y;
    int[] tab;
    ...
}
```

Possibilité d'utiliser le **même identificateur**
dans deux méthodes distinctes
pas de conflit, c'est la déclaration locale qui
est utilisée dans le corps de la méthode

```
double methode2(...) {
    double x;
    double y;
    double[] tab;
    ...
}
```

(Syntaxe JAVA : visibilité des variables)

- De manière générale
  - Variable visible à l'intérieur du bloc (ensembles des instructions entre { … }) où elle est définie

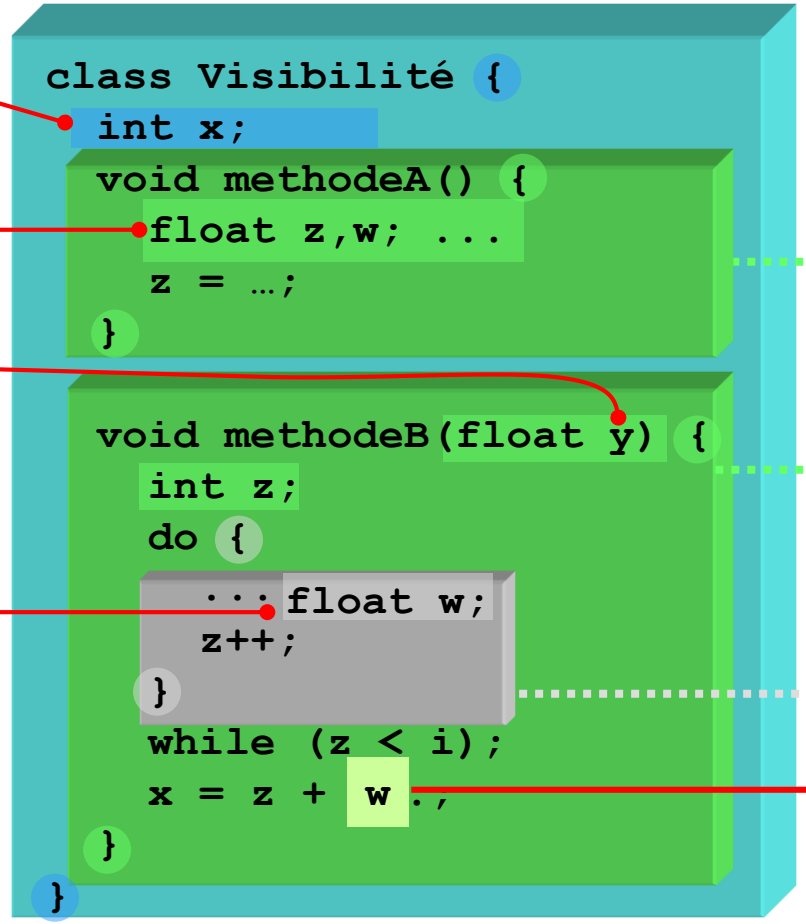
attribut (ou variable d'instance) accessible dans toute la classe

variables locales - accessibles dans la méthode où elles sont définies

paramètre, se comporte comme une variable locale

variable locale à un bloc accessible que dans le bloc où elle est définie

la variable peut être définie n'importe où dans le bloc, elle n'est accessible qu'après son point de déclaration



x : int z : float w : float

x : int y : float z : int

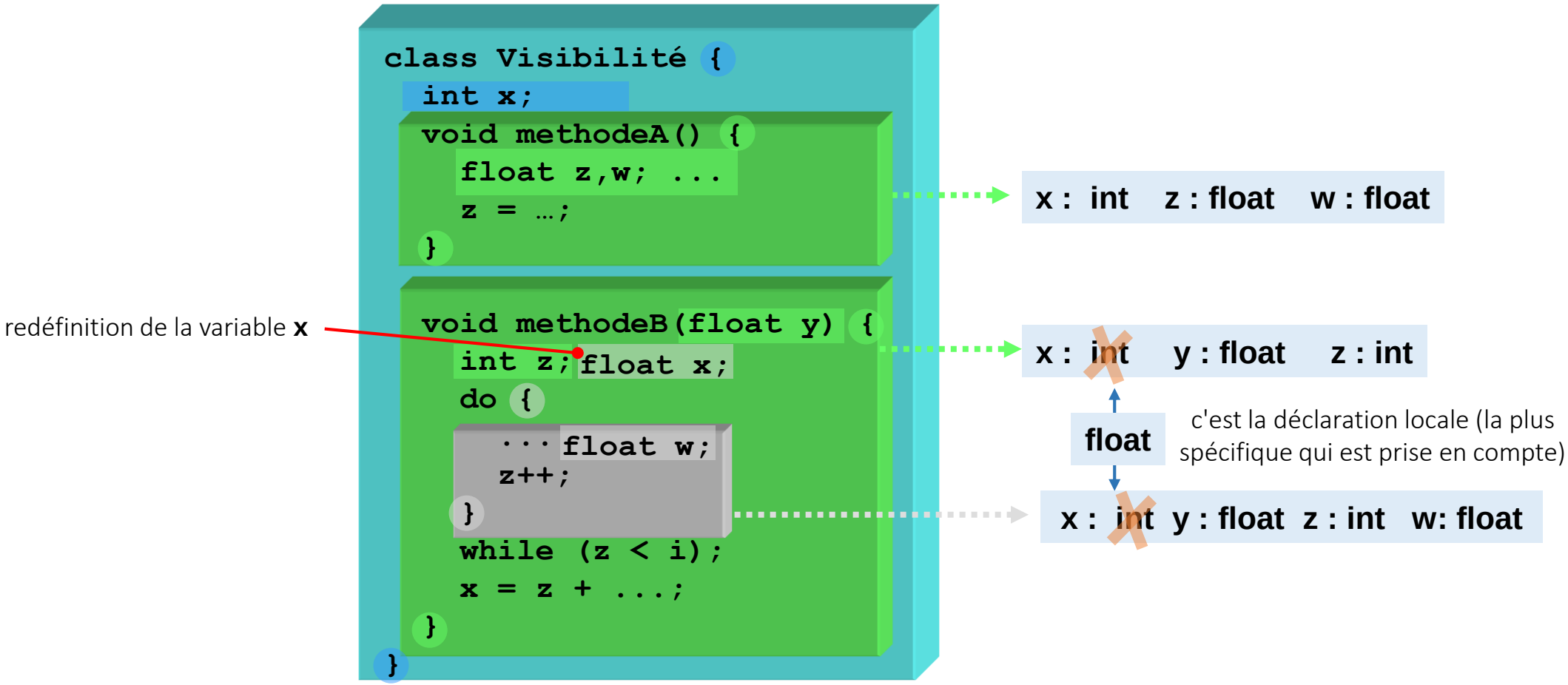
x : int y : float z : int w: float

Symbole non défini

erreur de compilation

(Syntaxe JAVA : visibilité des variables…)

- Attention !!
  - la redéfinition d'une variable masque la définition au niveau du bloc englobant.

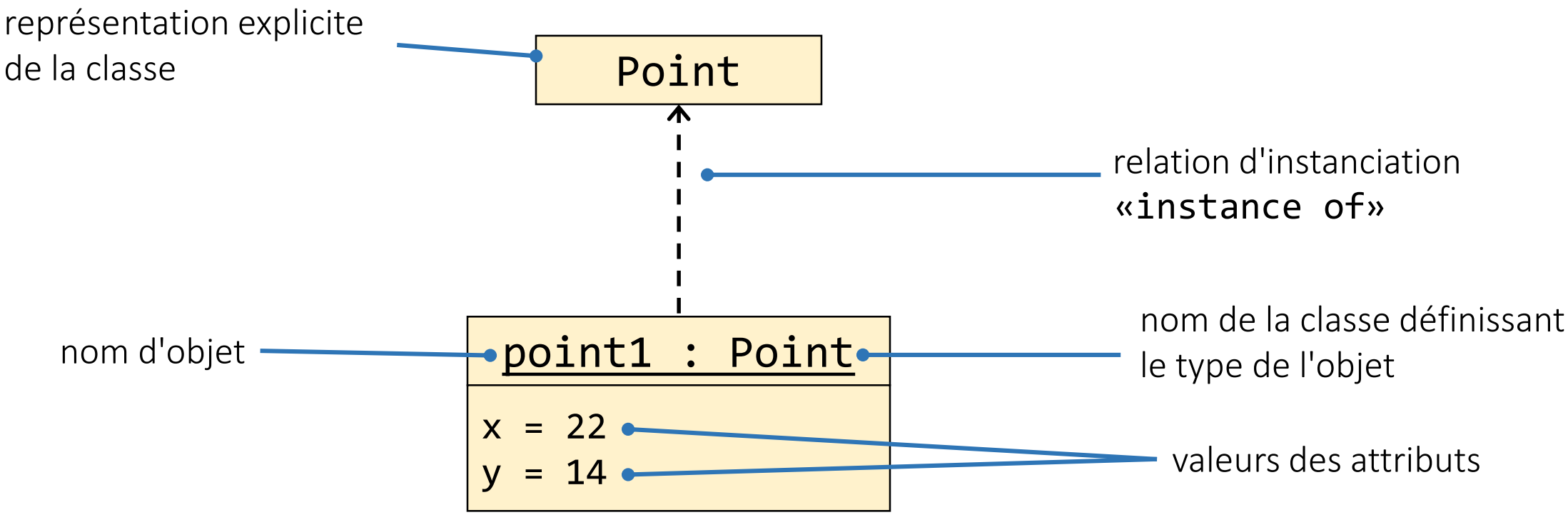


Objets

- Un objet est **instance** d'une (seule) classe :
  - *il se conforme à la description que celle-ci fournit,*
  - *il admet une valeur **(qui lui est propre)** pour chaque attribut déclaré dans la classe,*
  - *ces valeurs caractérisent l'**état** de l'objet*
  - *il est possible de lui appliquer toute opération (**méthode**) définie dans la classe (envoi de message)*

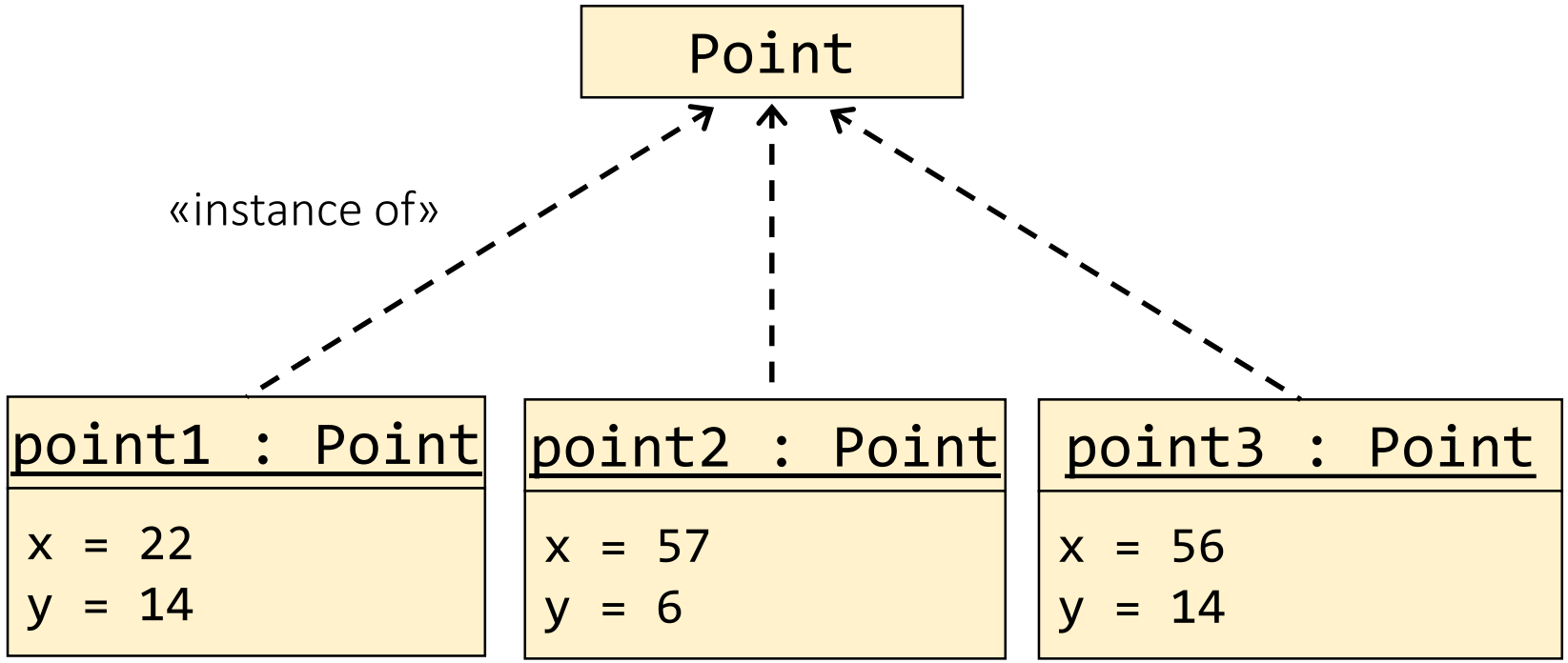
- Tout objet admet une identité qui le distingue pleinement des autres objets :
  - *il peut être nommé et être **référencé** par un nom*

- Notation d'un objet **point1**, instance de la classe **Point**



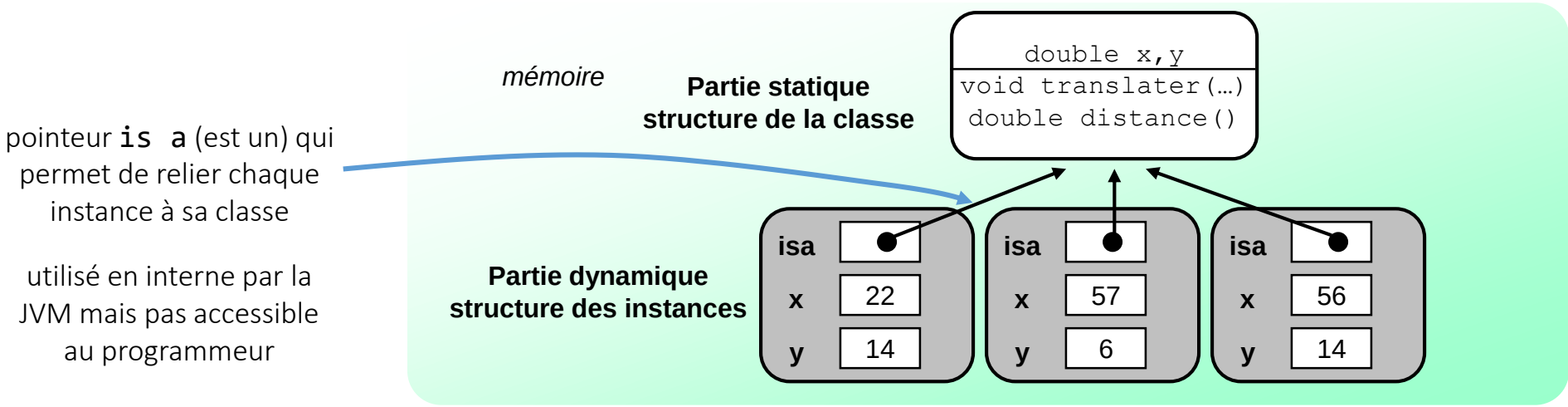
Objets

- Chaque objet point instance de la classe ***Point*** possède son propre ***x*** et son propre ***y***
- Différentes termes pour désigner x et y
  - attributs, propriétés de l'objet
  - variables d'instance, variables d'objet



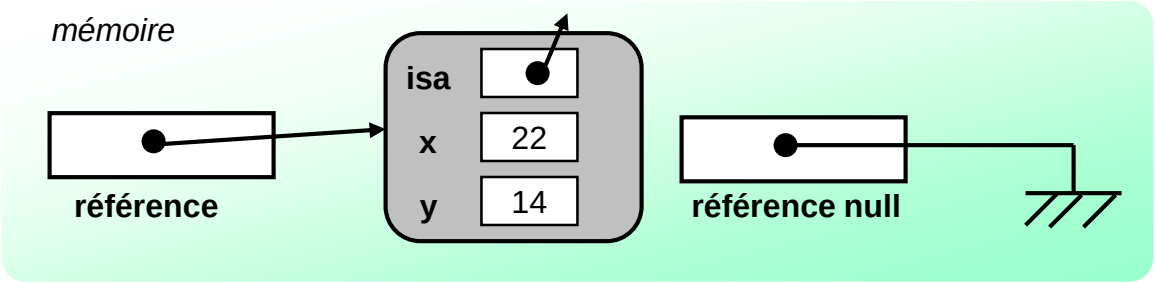
Structure des classes et objets en mémoire

- objet constitué d'une partie "**statique**" et d'une partie "**dynamique**"
  - *Partie statique (associée à la classe de l'objet) :*
    - *au niveau de la partie statique on retrouve ce qui est partagé par toutes les objets instances de la classe (par exemple les méthodes)*
    - *existe en un seul exemplaire et est commune à tous les objets instances de la classe*
  - *Partie dynamique (associée à l'objet lui même):*
    - *constituée d'un exemplaire de chaque variable d'instance définie par la classe.*
    - *chaque instance possède une partie dynamique qui lui est propre*
    - *varie durant la vie d'un objet*



Références

- Pour désigner des objets dans une classe (attributs ou variables dans le corps d ’une méthode) on utilise des variables d’un type particulier : les **références**
- Une référence contient l’**adresse** d’un objet
  - *pointeur vers la structure de données correspondant aux attributs (variables d’instance) propres à l’objet.*



- Une référence peut posséder la valeur `null`
  - *aucun objet n’est accessible par cette référence*
- Déclarer une référence ne crée pas d’objet
  - *une référence n’est pas un objet, c’est un nom pour accéder à un objet*

Références

déclaration en Java

- Déclarations de références

identificateur de classe

identificateur

```
Point p1;
Point p2, p3;
Cercle monCercle;
```

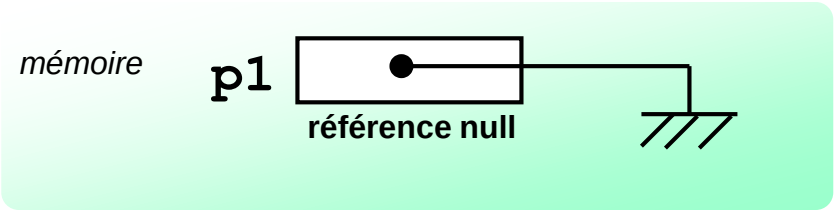
conventions d'écriture en Java
- Upper Camel Case (ou Pascal Case) pour les noms de classes
- Camel case pour les noms de variables ou de méthodes

- Par défaut à la déclaration une référence vaut null
  - *elle ne « pointe » sur aucun objet*

```
Point p1;
```

⟺

```
Point p1 = null;
```



Références en Java

- Les références java : des pointeurs *«Canada Dry™»*

Canada Dry est doré comme l'alcool, son nom sonne comme un nom d'alcool… mais ce n'est pas de l'alcool, et c'est pour cela que cela désaltère

https://www.youtube.com/watch?v=AK_EzfM5_38

- ***Comme un pointeur*** une référence contient l'adresse d'une structure
- Mais ***à la différence des pointeurs*** la seule opération autorisée sur les références est l'**affectation** d'une référence de même type*

* en réalité ce n'est pas aussi strict, à une référence d'un type donné il est possible d'affecter une référence d'un autre type à condition que les types soient compatibles (on en reparlera lors du cours sur l'héritage)

```
Point p1;
Point p2;
Cercle c1;
```

```
p1 = p2;
```

```
p1 = c1;
```

```
p1 = &p2;
p2 += *p1 +3;
```

sources d'erreurs en C, C++

Segmentation fault
Core dump

erreurs détectées dès la compilation

Création d'Objets

- La création d 'un objet à partir d'une classe est appelée **instanciation**. L'objet créé est une **instance** de la classe.
- Instanciation se décompose en trois phases :
  - *1 : **obtention de l 'espace mémoire** nécessaire à la partie dynamique de l'objet et initialisation des attributs en mémoire (à l'image d'une structure)*
  - *2 : **appel d'un des constructeurs** définis dans la classe (ou au constructeur par défaut si pas de constructeur explicitement défini. On en reparlera plus tard :-)*
  - *3 : renvoi d'une **référence sur l'objet** (son identité) maintenant créé et initialisé.*

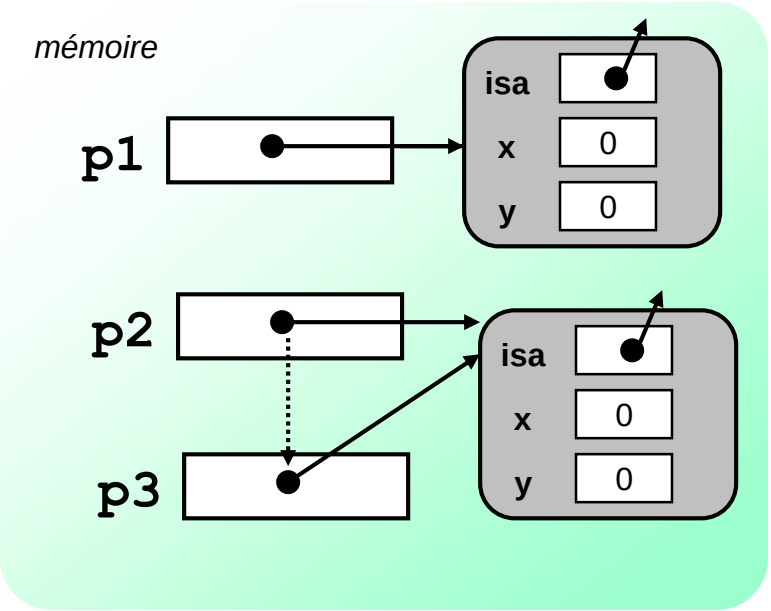
Création d'Objets

Instanciation en Java

`new NomDeClasse(<liste d'arguments>)`

- les constructeurs sont des méthodes particulières qui ont le **même nom que la classe** (casse comprise) et n'ont pas de déclaration de type de retour
- si dans le code de la classe il n'y a pas de définition explicite de constructeur, il existe un constructeur par défaut
  - *sans paramètres*
  - *réduit à phase 1 (allocation mémoire) + phase 3 (retour adresse)*
  - *inexistant si un autre constructeur existe*

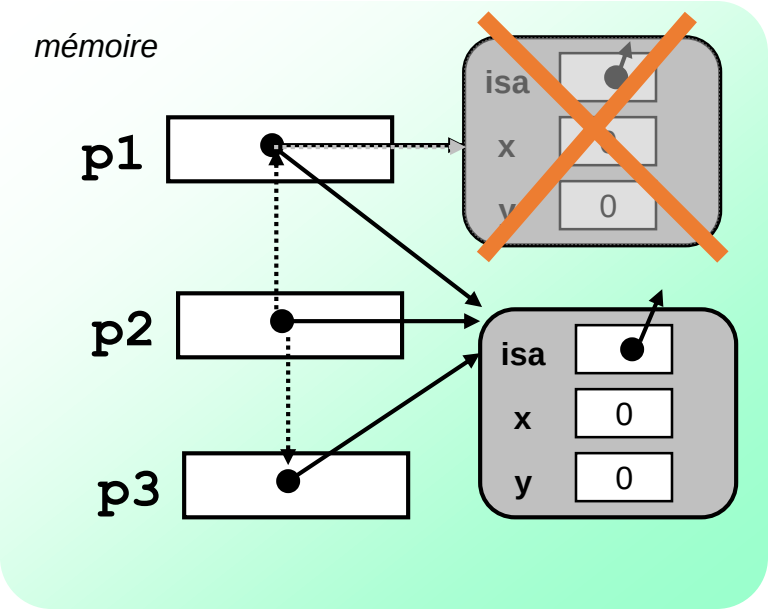
```
Point p1;
p1 = new Point();
Point p2 = new Point();
Point p3 = p2;
```



Création d'Objets

- L'instanciation provoque une allocation dynamique de la mémoire
- En Java le programmeur n'a pas à se soucier de la gestion mémoire
  - *Si un objet n'est plus référencé (plus accessible au travers d'une référence), la mémoire qui lui était allouée est* ***automatiquement " libérée "*** *(le « garbage collector » la récupérera en temps voulu).*
  - *Attention : destruction* ***asynchrone*** *(car gérée par un thread)*
    Aucune garantie du moment où la destruction aura lieu (sauf en fin de programme! Ou appel explicite au garbage collector)

```
Point p1;
p1 = new Point();
Point p2 = new Point();
Point p3 = p2;
p1 = p2;
```

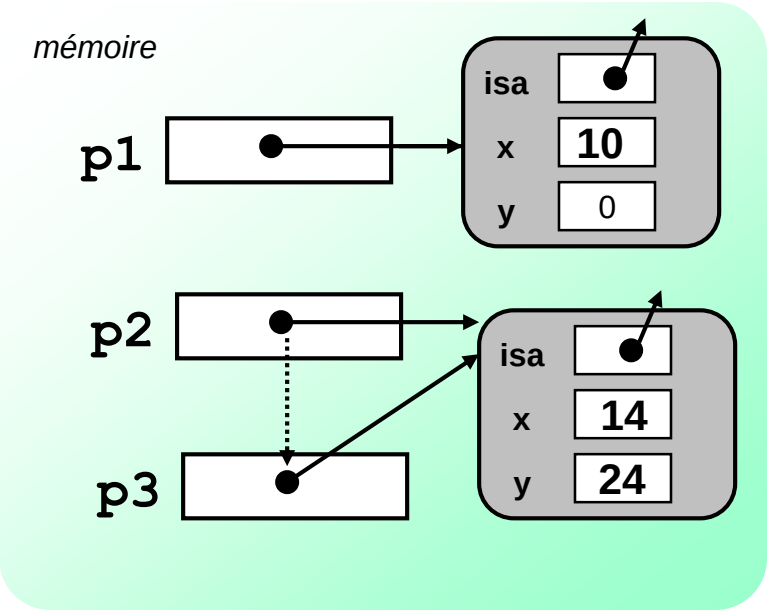


Accès aux attributs d'un objet (en Java)

- pour accéder aux attributs d'un objet on utilise une notation pointée :

```
referenceObjet.nomDeVariableDinstance
```

```
Point p1;
p1 = new Point();
Point p2 = new Point();
Point p3 = p2;
p1.x = 10;
p2.x = 14;
p3.y = p1.x + p2.x;
```



Envoi de messages

- pour "demander" à un objet d'effectuer une opération (exécuter l'une de ses méthodes) il faut lui **envoyer un message**

- un message est composé de trois parties
  - *une **référence** permettant de désigner l'objet à qui le message est envoyé*
  - *le **nom de la méthode** à exécuter (cette méthode doit bien entendu être définie dans la classe de l'objet)*
  - *les éventuels **paramètres** de la méthode*

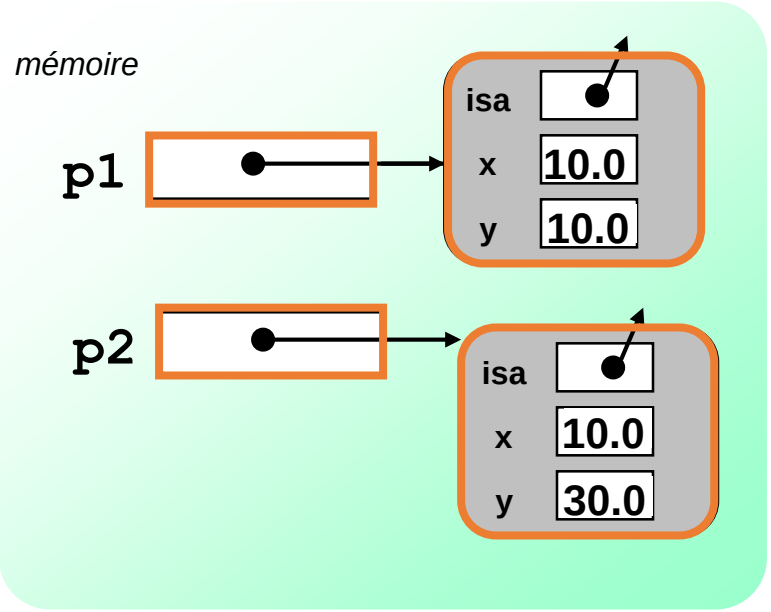
- envoi de message similaire à un appel de fonction
  - *les instructions définies dans la méthode sont exécutées (elles s'appliquent sur les attributs de l'objet récepteur du message)*
  - *puis le contrôle est retourné au programme appelant*

Envoi de messages

exemple en JAVA

référenceObjet.nomDeMethode(<liste d'arguments>)

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

   void translater(double dx, double dy) {
     x += dx; y += dy;
   }

   double distance() {
     return Math.sqrt(x*x+y*y);
   }
} // Point
```




```
Point p1 = new Point();
Point p2 = new Point();

p1.translater(10.0,10.0);

p2.translater(p1.x,3.0 * p1.y);

System.out.println("distance de p1 à origine "
          + p1.distance());
```

si la méthode ne possède pas de paramètres, la liste d'arguments est vide, mais comme en langage C les parenthèses demeurent

Envoi de messages

Paramètres des méthodes

- un paramètre d'une méthode peut être :
  - *Une variable de type simple*
  - *Une référence typée par n'importe quelle classe (connue dans le contexte de compilation)*
    - *exemple : ajout à la classe* **`Point`** *d'une méthode qui permet de placer le* **`Point`** *au même endroit qu'un autre* **`Point`**.

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

   void translater(double dx, double dy){
      x += dx;
      y += dy;
   }

   double distance() {
     return Math.sqrt(x * x + y * y);
   }

}
```

Déclaration de la méthode

```
/**
* Place le Point (qui reçoit le message) au même endroit qu'un autre Point
* @param p le Point servant à définir la nouvelle position du Point
*/
void placerA(Point p) {
    x = p.x;
    y = p.y
}
```

Utilisation de la méthode

```
Point p1 = new Point();
Point p2 = new Point();
...
p1.placerA(p2); // p1 prend les mêmes coordonnées que celle de p2
```

Envoi de messages

Passage des paramètres

- Le passage de paramètres lors de l'envoi de message est un **passage par valeur**.
  - *À l'exécution le paramètre formel défini dans la signature de la méthode correspond à une variable locale au bloc de la méthode*
  - *Elle est initialisée avec la valeur de l'expression définie par le paramètre effectif.*

```
00  class Point {
..     ...
20     void foo(int x, Point p) {
21       ...
22       p.translater(10,10);
23       x = x + 10;
24       p = new Point();
25       p.translater(x,10);
26       ...
27     }
28  }
```

```
00  Point p1 = new Point();
01  Point p2 = new Point();
02  p2.x = 15; p2.y = 11;
03  int x = 20;
04  p1.foo(x,p2);
05  System.out.printl("x " + x);
06  System.out.println("p2.x " + p2.x);
07  System.out.println("p2.y " + p2.y);
```

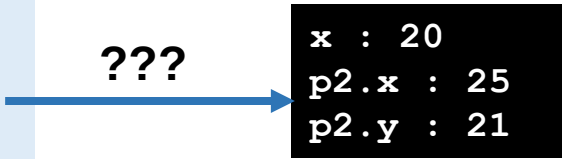
Envoi de messages

Passage des paramètres

- Le passage de paramètres lors de l'envoi de message est un **passage par valeur**.
  - *À l'exécution le paramètre formel défini dans la signature de la méthode correspond à une variable locale au bloc de la méthode*
  - *Elle est initialisée avec la valeur de l'expression définie par le paramètre effectif.*

```
00  class Point {
..     ...
20     void foo(int x, Point p) {
21       ...
22       p.translater(10,10);
23       x = x + 10;
24       p = new Point();
25       p.translater(x,10);
26       ...
27     }
28  }
```

```
00  Point p1 = new Point();
01  Point p2 = new Point();
02  p2.x = 15; p2.y = 11;
03  int x = 20;
04  p1.foo(x,p2);
05  System.out.printl("x " + x);
06  System.out.println("p2.x " + p2.x);
07  System.out.println("p2.y " + p2.y);
```



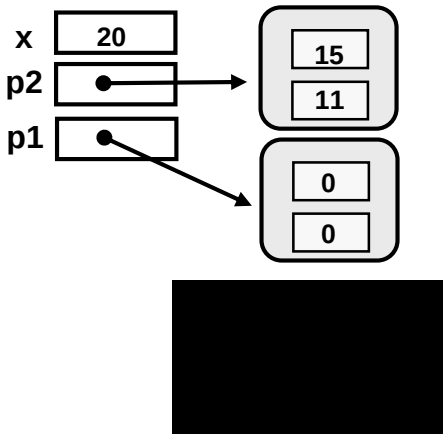
Envoi de messages

Passage des paramètres

- Le passage de paramètres lors de l'envoi de message est un **passage par valeur**.
  - *À l'exécution le paramètre formel défini dans la signature de la méthode correspond à une variable locale au bloc de la méthode*
  - *Elle est initialisée avec la valeur de l'expression définie par le paramètre effectif.*

```
00  class Point {
..     ...
20     void foo(int x, Point p) {
21       ...
22       p.translater(10,10);
23       x = x + 10;
24       p = new Point();
25       p.translater(x,10);
26       ...
27     }
28  }
```

```
00  Point p1 = new Point();
01  Point p2 = new Point();
02  p2.x = 15; p2.y = 11;
03  int x = 20;
04  p1.foo(x,p2);
05  System.out.println("x " + x);
06  System.out.println("p2.x " + p2.x);
07  System.out.println("p2.y " + p2.y);
```



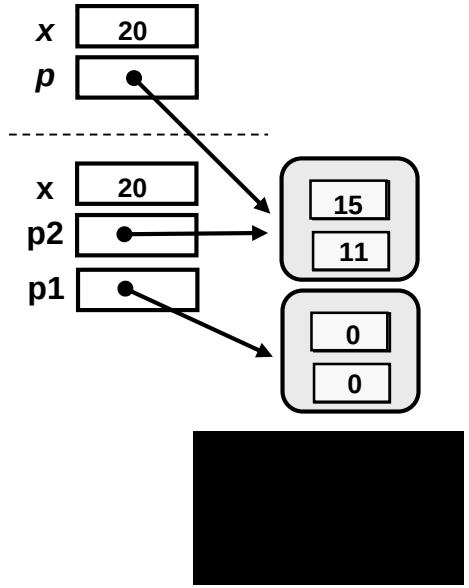
Envoi de messages

Passage des paramètres

- Le passage de paramètres lors de l'envoi de message est un **passage par valeur**.
  - *À l'exécution le paramètre formel défini dans la signature de la méthode correspond à une variable locale au bloc de la méthode*
  - *Elle est initialisée avec la valeur de l'expression définie par le paramètre effectif.*

```
00  class Point {
..      ...
20      void foo(int x, Point p) {
21        ...
22        p.translater(10,10);
23        x = x + 10;
24        p = new Point();
25        p.translater(x,10);
26        ...
27      }
28  }
```

```
00  Point p1 = new Point();
01  Point p2 = new Point();
02  p2.x = 15; p2.y = 11;
03  int x = 20;
04  p1.foo(x,p2);
05  System.out.printl("x " + x);
06  System.out.println("p2.x " + p2.x);
07  System.out.println("p2.y " + p2.y);
```



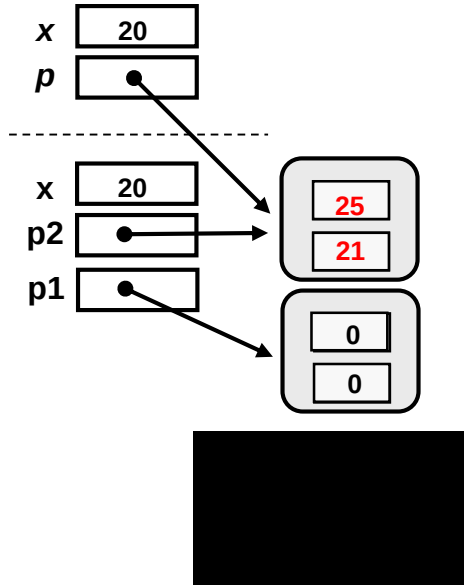
Envoi de messages

Passage des paramètres

- Le passage de paramètres lors de l'envoi de message est un **passage par valeur**.
  - *À l'exécution le paramètre formel défini dans la signature de la méthode correspond à une variable locale au bloc de la méthode*
  - *Elle est initialisée avec la valeur de l'expression définie par le paramètre effectif.*

```
00  class Point {
..     ...
20     void foo(int x, Point p) {
21       ...
22       p.translater(10,10);
23       x = x + 10;
24       p = new Point();
25       p.translater(x,10);
26       ...
27     }
28  }
```

```
00  Point p1 = new Point();
01  Point p2 = new Point();
02  p2.x = 15; p2.y = 11;
03  int x = 20;
04  p1.foo(x,p2);
05  System.out.println("x " + x);
06  System.out.println("p2.x " + p2.x);
07  System.out.println("p2.y " + p2.y);
```



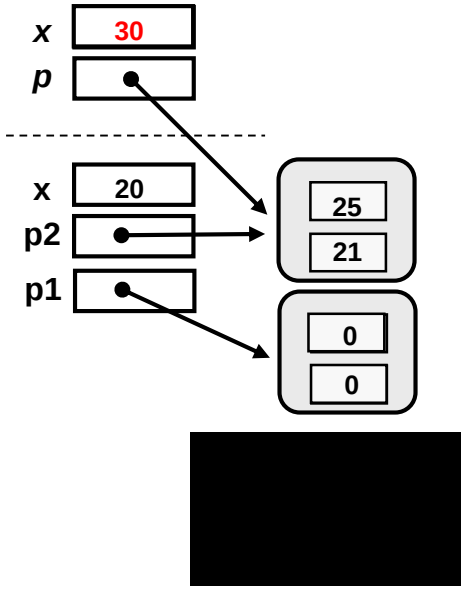
Envoi de messages

Passage des paramètres

- Le passage de paramètres lors de l'envoi de message est un **passage par valeur**.
  - *À l'exécution le paramètre formel défini dans la signature de la méthode correspond à une variable locale au bloc de la méthode*
  - *Elle est initialisée avec la valeur de l'expression définie par le paramètre effectif.*

```
00  class Point {
..     ...
20     void foo(int x, Point p) {
21       ...
22       p.translater(10,10);
23       x = x + 10;
24       p = new Point();
25       p.translater(x,10);
26       ...
27     }
28  }
```

```
00  Point p1 = new Point();
01  Point p2 = new Point();
02  p2.x = 15; p2.y = 11;
03  int x = 20;
04  p1.foo(x,p2);
05  System.out.println("x " + x);
06  System.out.println("p2.x " + p2.x);
07  System.out.println("p2.y " + p2.y);
```



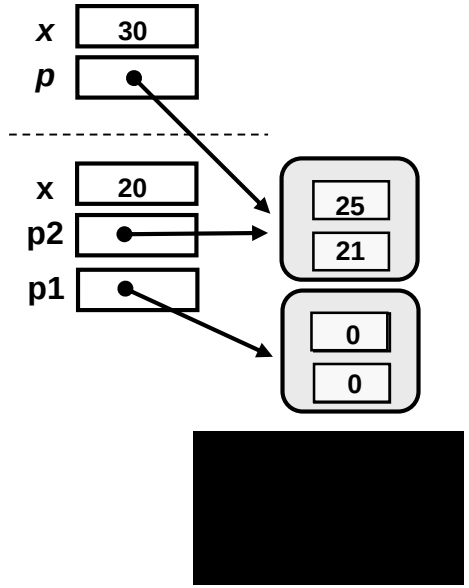
Envoi de messages

Passage des paramètres

- Le passage de paramètres lors de l'envoi de message est un **passage par valeur**.
  - *À l'exécution le paramètre formel défini dans la signature de la méthode correspond à une variable locale au bloc de la méthode*
  - *Elle est initialisée avec la valeur de l'expression définie par le paramètre effectif.*

```
00  class Point {
..     ...
20     void foo(int x, Point p) {
21       ...
22       p.translater(10,10);
23       x = x + 10;
24       p = new Point();
25       p.translater(10,10);
26       ...
27     }
28  }
```

```
00  Point p1 = new Point();
01  Point p2 = new Point();
02  p2.x = 15; p2.y = 11;
03  int x = 20;
04  p1.foo(x,p2);
05  System.out.printl("x " + x);
06  System.out.println("p2.x " + p2.x);
07  System.out.println("p2.y " + p2.y);
```



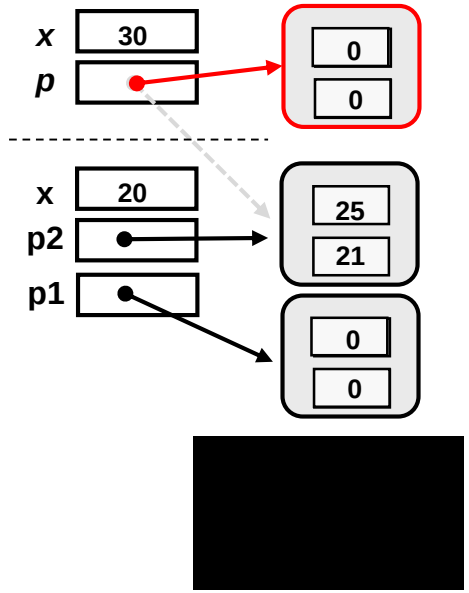
Envoi de messages

Passage des paramètres

- Le passage de paramètres lors de l'envoi de message est un **passage par valeur**.
  - *À l'exécution le paramètre formel défini dans la signature de la méthode correspond à une variable locale au bloc de la méthode*
  - *Elle est initialisée avec la valeur de l'expression définie par le paramètre effectif.*

```
00  class Point {
..     ...
20     void foo(int x, Point p) {
21       ...
22       p.translater(10,10);
23       x = x + 10;
24       p = new Point();
25       p.translater(x,10);
26       ...
27     }
28  }
```

```
00  Point p1 = new Point();
01  Point p2 = new Point();
02  p2.x = 15; p2.y = 11;
03  int x = 20;
04  p1.foo(x,p2);
05  System.out.printl("x " + x);
06  System.out.println("p2.x " + p2.x);
07  System.out.println("p2.y " + p2.y);
```



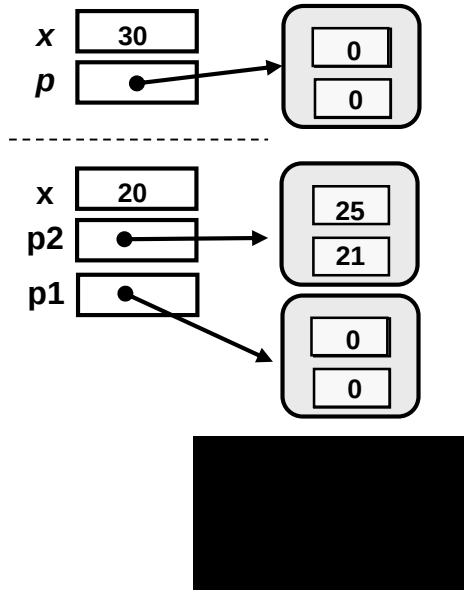
Envoi de messages

Passage des paramètres

- Le passage de paramètres lors de l'envoi de message est un **passage par valeur**.
  - *À l'exécution le paramètre formel défini dans la signature de la méthode correspond à une variable locale au bloc de la méthode*
  - *Elle est initialisée avec la valeur de l'expression définie par le paramètre effectif.*

```
00  class Point {
..     ...
20     void foo(int x, Point p) {
21       ...
22       p.translater(10,10);
23       x = x + 10;
24       p = new Point();
25       p.translater(x,10);
26       ...
27     }
28  }
```

```
00  Point p1 = new Point();
01  Point p2 = new Point();
02  p2.x = 15; p2.y = 11;
03  int x = 20;
04  p1.foo(x,p2);
05  System.out.printl("x " + x);
06  System.out.println("p2.x " + p2.x);
07  System.out.println("p2.y " + p2.y);
```



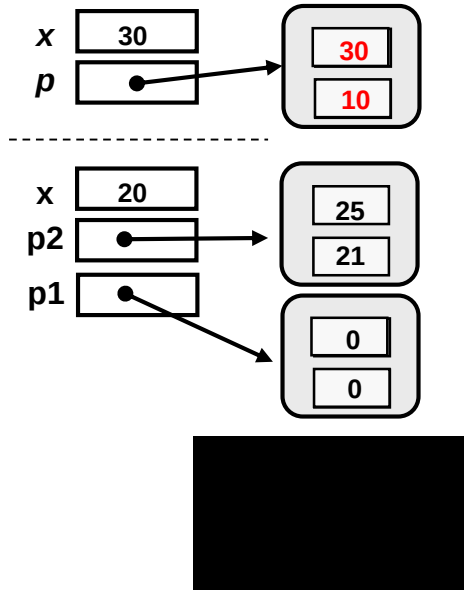
Envoi de messages

Passage des paramètres

- Le passage de paramètres lors de l'envoi de message est un **passage par valeur**.
  - *À l'exécution le paramètre formel défini dans la signature de la méthode correspond à une variable locale au bloc de la méthode*
  - *Elle est initialisée avec la valeur de l'expression définie par le paramètre effectif.*

```
00  class Point {
..     ...
20     void foo(int x, Point p) {
21       ...
22       p.translater(10,10);
23       x = x + 10;
24       p = new Point();
25       p.translater(x,10);
26       ...
27     }
28  }
```

```
00  Point p1 = new Point();
01  Point p2 = new Point();
02  p2.x = 15; p2.y = 11;
03  int x = 20;
04  p1.foo(x,p2);
05  System.out.println("x " + x);
06  System.out.println("p2.x " + p2.x);
07  System.out.println("p2.y " + p2.y);
```



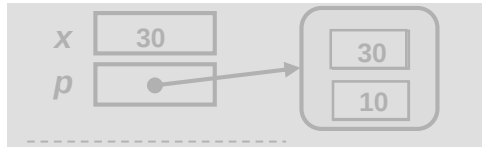
Envoi de messages

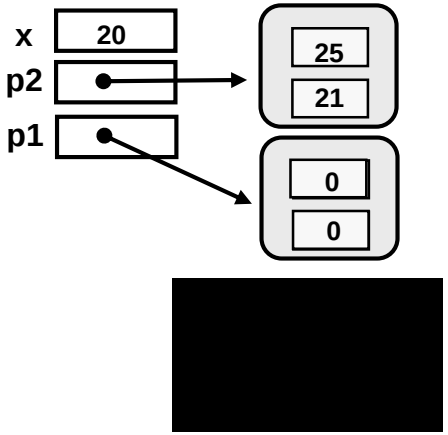
Passage des paramètres

- Le passage de paramètres lors de l'envoi de message est un **passage par valeur**.
  - *À l'exécution le paramètre formel défini dans la signature de la méthode correspond à une variable locale au bloc de la méthode*
  - *Elle est initialisée avec la valeur de l'expression définie par le paramètre effectif.*

```
00  class Point {
..     ...
20     void foo(int x, Point p) {
21       ...
22       p.translater(10,10);
23       x = x + 10;
24       p = new Point();
25       p.translater(x,10);
26       ...
27     }
28  }
```

```
00  Point p1 = new Point();
01  Point p2 = new Point();
02  p2.x = 15; p2.y = 11;
03  int x = 20;
04  p1.foo(x,p2);
05  System.out.println("x " + x);
06  System.out.println("p2.x " + p2.x);
07  System.out.println("p2.y " + p2.y);
```





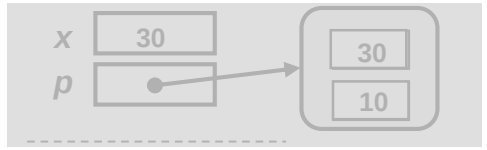
Envoi de messages

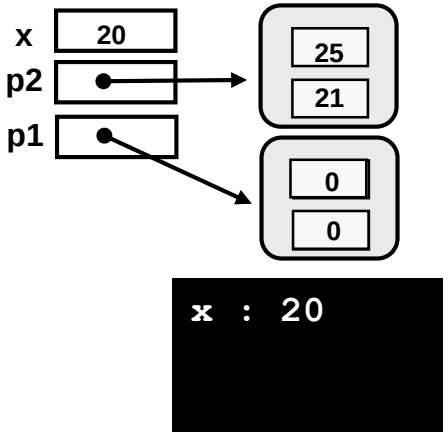
Passage des paramètres

- Le passage de paramètres lors de l'envoi de message est un **passage par valeur**.
  - *À l'exécution le paramètre formel défini dans la signature de la méthode correspond à une variable locale au bloc de la méthode*
  - *Elle est initialisée avec la valeur de l'expression définie par le paramètre effectif.*

```
00  class Point {
..      ...
20      void foo(int x, Point p) {
21        ...
22        p.translater(10,10);
23        x = x + 10;
24        p = new Point();
25        p.translater(x,10);
26        ...
27      }
28  }
```

```
00  Point p1 = new Point();
01  Point p2 = new Point();
02  p2.x = 15; p2.y = 11;
03  int x = 20;
04  p1.foo(x,p2);
05  System.out.printl("x " + x);
06  System.out.println("p2.x " + p2.x);
07  System.out.println("p2.y " + p2.y);
```





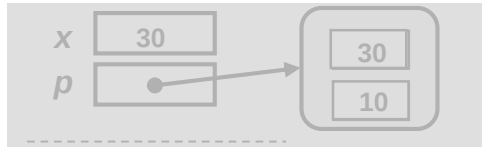
Envoi de messages

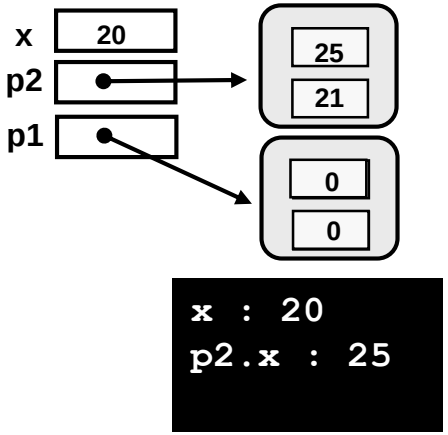
Passage des paramètres

- Le passage de paramètres lors de l'envoi de message est un **passage par valeur**.
  - *À l'exécution le paramètre formel défini dans la signature de la méthode correspond à une variable locale au bloc de la méthode*
  - *Elle est initialisée avec la valeur de l'expression définie par le paramètre effectif.*

```
00  class Point {
..      ...
20      void foo(int x, Point p) {
21        ...
22        p.translater(10,10);
23        x = x + 10;
24        p = new Point();
25        p.translater(x,10);
26        ...
27      }
28  }
```

```
00  Point p1 = new Point();
01  Point p2 = new Point();
02  p2.x = 15; p2.y = 11;
03  int x = 20;
04  p1.foo(x,p2);
05  System.out.println("x " + x);
06  System.out.println("p2.x " + p2.x);
07  System.out.println("p2.y " + p2.y);
```





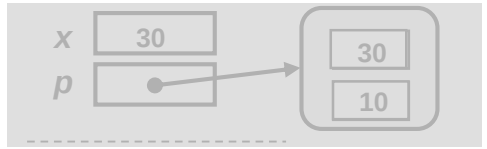
Envoi de messages

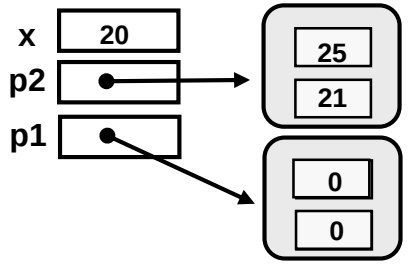
Passage des paramètres

- Le passage de paramètres lors de l'envoi de message est un **passage par valeur**.
  - *À l'exécution le paramètre formel défini dans la signature de la méthode correspond à une variable locale au bloc de la méthode*
  - *Elle est initialisée avec la valeur de l'expression définie par le paramètre effectif.*

```
00  class Point {
..     ...
20     void foo(int x, Point p) {
21       ...
22       p.translater(10,10);
23       x = x + 10;
24       p = new Point();
25       p.translater(x,10);
26       ...
27     }
28  }
```

```
00  Point p1 = new Point();
01  Point p2 = new Point();
02  p2.x = 15; p2.y = 11;
03  int x = 20;
04  p1.foo(x,p2);
05  System.out.println("x " + x);
06  System.out.println("p2.x " + p2.x);
07  System.out.println("p2.y " + p2.y);
```






```
x : 20
p2.x : 25
p2.y : 21
```

L'objet « courant »

Le mot clé `this` (en Java)

- dans un message l'accent est mis sur l'objet (et non pas sur l'appel de fonction)
  - *en JAVA (et de manière plus générale en POO) on écrit :*

```
d1 = p1.distance(); d2 = p2.distance();
```

  - *en C on aurait probablement écrit :*

```
d1 = distance(p1); d2 = distance(p2);
```

- l'objet qui reçoit un message est implicitement passé comme argument à la méthode invoquée

- cet argument implicite défini par le mot clé **`this`** (`self, current` dans d'autres langages)
  - *une référence particulière*
  - *désigne **l'objet courant** :*
    - *objet récepteur du message, auquel s'appliquent les instructions du corps de la méthode où **`this`** est utilisé*
  - *peut être utilisé pour rendre explicite l'accès aux propres attributs et méthodes définies dans la classe*

L'objet « courant »

this et variables d'instance

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

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

  double distance() {
      return Math.sqrt(x * x + y * y);
  }    ⇔ return Math.sqrt(this.x * this.x + this.y * this.y;


  void placerA(Point p){
      x = p.x;
      y = p.y;
  }     ⇔ this.x = p.x; this.y= p.y;


  void placerAuxCoordonnées(double x, double y){
      this.x = x;
      this.y = y;
  }
}
```

dans le corps d'une méthode lorsque un attribut est utilisé, c'est un attribut de l'objet courant (l'utilisation de **this** est implicite)

dans le cas ou un attribut est masqué (par un paramètre ou une variable locale) **this** doit être explicitement utilisé pour lever les ambiguïtés

```
void placerAuxCoordonnées(double x, double y){
   x = x;
   y = y;
}
```

serait sans effet sur l'objet

L'objet « courant »

this et envoi de message

- Utilisation de **this** dans le code d'une classe pour invoquer l'une des méthodes qu'elle définit (récursivité possible)

*exemple : ajout à la classe **Point** d'une méthode qui permet savoir si le **Point** est plus proche de l'origine qu'un autre **Point**.*

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

   void translater(double dx, double dy){
      x += dx;
      y += dy;
   }

   double distance() {
     return Math.sqrt(x * x + y * y);
   }

   ...
}
```

```
/**
* Teste si le Point (qui reçoit le message) est plus proche de l'origine qu'un
* autre Point.
* @param p le Point avec lequel le Point recevant le message doit être comparé
* @return true si le point recevant le message est plus proche de l'origine
* que p, false sinon.
*/
```

```
boolean plusProcheDeOrigineQue ( Point p ) {
   return this.distance() < p.distance();
}
```

⇔ *return distance() < p.distance();*

L'objet qui reçoit le message **plusProcheDeOrigineQue** invoque sur **lui-même** la méthode **distance** (il s'envoie un message **distance**) **this** n'est pas indispensable

l'ordre de définition des méthodes n'a pas d 'importance. **plusProcheDeOrigineQue** pourrait être définie avant **distance**

```
Point p1 = new Point();
Point p2 = new Point();
...
if (p1.plusProcheOrigineQue(p2))
   System.out.println("p1 est plus proche de l'origine que p2");
else
   System.out.println("p2 est plus proche de l'origine que p1");
```

lorsque la méthode **plusProcheOrigineQue()** est exécutée
this a comme valeur la valeur de **p1**
p a comme valeur la valeur de **p2**

L'objet « courant »

autre utilisation de this

Quand l'objet récepteur du message doit se passer en paramètre d'une méthode ou sa référence doit être retournée par la méthode

VisageRond.java

```
class VisageRond {
   Dessin d;
   int x, int y;
   ...
   boolean bordAtteint(){
     if ( x > d.largeur() ... )
        return false;
     else
        ...
   }
   ...
   void setDessin(Dessin d)
     this.d = d;
   }
}
```

3

AppliVisage.java

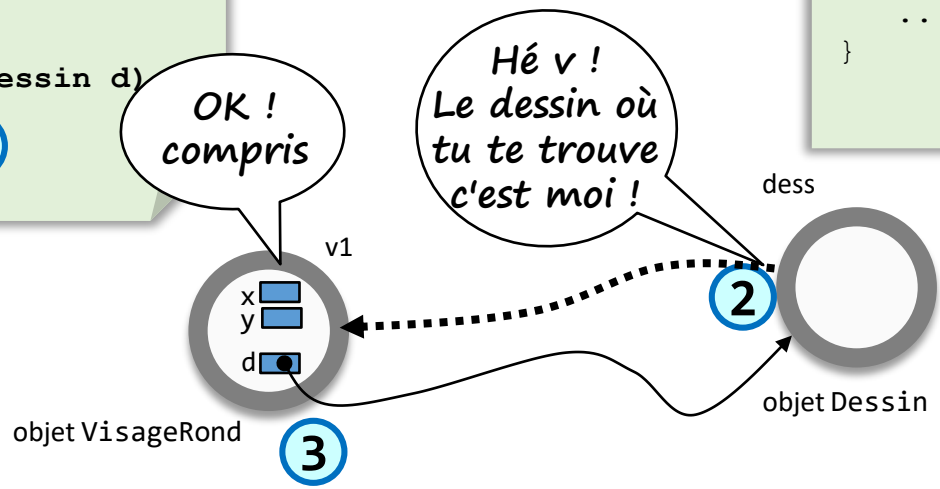
```
...
Dessin dess=new Dessin();
VisageRond v1 =
     new VisageRond();
...

dess.ajouter(v1);
```

1

Dessin.java

```
class Dessin {

   ...

   void ajouter(VisageRond v){
      ...
      v.setDessin(this);
      ...
   }
   ...
}
```

2



Encapsulation

- accès direct aux variables d'un objet possible en JAVA

```
Point p1 = new Point();
p1.x = 15; p2.y = 11;
System.out.println("x : " + p1.x
            + " y : " + p1.y)
```

- **mais ...** n'est pas recommandé car contraire au **principe d'encapsulation**
  - *les données d'un objet doivent être privées (c'est à dire protégées et accessibles (et surtout modifiables) qu'au travers de méthodes prévues à cet effet).*

- en JAVA, possible lors de leur définition d'agir sur la **visibilité** (accessibilité) des **membres** (attributs et méthodes) d'une classe vis à vis des autres classes

- plusieurs niveaux de visibilité peuvent être définis en précédant la déclaration de chaque attribut, méthode ou constructeur d'un modificateur (**`private`**, **`public`**, **`protected`**, **`-`** (package))

Encapsulation

Visibilité des membres d'une classe (en Java)

	`public`	`private`
classe	La classe peut être utilisée dans n'importe quelle autre classe	interdit
attribut	Attribut accessible directement depuis code de n'importe quelle classe	Attribut accessible uniquement dans le code de la classe qui le définit
méthode	Méthode pouvant être invoquée depuis code de n'importe quelle classe	Méthode utilisable uniquement dans le code de la classe qui la définit

- *on reverra les autres niveaux de visibilité (-, `protected`) en détail lorsque les notions d'héritage et de packages auront été abordées*

Encapsulation

Visibilité des attributs (en Java)

- les attributs déclarés comme privées (`private`) sont totalement protégés
  - *ils ne sont plus directement accessibles depuis le code d'une autre classe*

code écrit en dehors de la classe `Point`

```
Point p1 = new Point();
...                          p1.getX()   p1.getY()
System.out.println(p1.x + " " + p1.y);
...
p1.x = 10;      p1.setX(10);
p1.y = 10;      p1.setY(p1.getX()+ 14);
```

pour accéder à leur valeur il faut passer par une méthode de type fonction (getter)

pour les modifier il faut passer par une méthode de type procédure (setter)

```
public class Point {
   private double x;
   private double y;
   public void translater(int dx, int dy) {
        x += dx; y += dy;
   }
   public double distance() {
        return Math.sqrt(x*x+y*y);
   }
   public double getX(){
        return x;
   }
   … idem pour y
   public void setX(double x){
        this.x = x;
   }
   … idem pour y
}
```

getters et setters : standard JavaBeans peuvent être générés automatiquement par IDE

Encapsulation

attributs privés

- les attributs déclarés comme privées (`private`) sont totalement protégés (principe d'encapsulation)
  - *ne sont plus directement accessibles depuis le code d'une autre classe*
  - *cela n'empêche pas de pouvoir s'en servir dans le code de la classe où ils sont définis*

Point.java

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

   ...
   /**
    *  Place le point au point spécifié
    *  @param p le
    */
   public void placerA(Point p) {
      this.x =  p.x ;
      this.y =  p.y;
   }
}
```

Une autre clase

```
...
Point p1 = new Point();
p1.x = 10;
p1.y = 10;
Point p2 = new Point();
p2.x = p1.x;
p2.y = p1.x + p1.y;
...
```

Encapsulation

Méthodes privées

- Une classe peut définir des méthodes privées à usage interne

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

   // constructeurs
   public Point(double dx, double dy){
         ...
   }
   // méthodes
  private double distance() {
         return Math.sqrt(x*x+y*y);
   }

   public boolean plusProcheDeOrigineQue(Point p){

         return this.distance() < p.distance();

   }
    ...
}
```

une méthode privée ne peut plus être invoquée en dehors du code de la classe où elle est définie

```
public class X {
  ...

  Point p=new Point()
  ...

  ... p.distance() ...

  ...
}
```

Encapsulation

Intérêt

- Accès au données ne se fait qu'au travers des méthodes.

  Un objet ne peut être utilisé que de la manière prévue à la conception de sa classe, sans risque d'utilisation incohérente → **Robustesse du code.**

fichier `Pixel.java`

```
/**
 * représentation d'un point de l'écran
 */
public class Pixel {
   // représentation en coordonnées cartésiennes
   private int x;   // 0 <= x < 1024
   private int y;   // 0 <= y < 780

   public int getX() {
     return x;
   }

   public void translater(int dx,int dy) {
     if ( ((x + dx) < 1024) && ((x + dx) >= 0) )
        x = x +dx;
     if ( ((y + dy) < 780) && ((y + dy) >= 0) )
          y = x + dy;
   }
   ...

} // Pixel
```

Code utilisant la classe **`Pixel`**

```
Pixel p1 = new Pixel();
p1.translater(100,100);
p1.translater(1000,-300);
...
p1.x = -100;
```

Impossible d'avoir un pixel dans un état incohérent
`( x < 0 ou x >= 1024 ou y < 0 ou y >= 780)`

Encapsulation

Intérêt

- Masquer l'implémentation → **facilite évolution du logiciel**

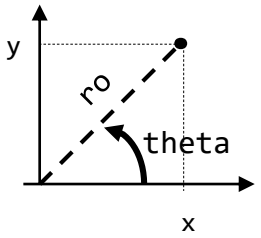
```
/** représentation d'un point du plan
*/
public class Point {
   // représentation en coordonnées cartésiennes
   private double x;
   private double y;

   public double distance() {
      return Math.sqrt(x*x+y*y);
   }

   public double getX() {
      return x;
   }

   public double getY() {
      return y;
   }

   public void translater(double dx, double dy) {
      ...
   }

   ...
} // Point
```

```
// représentation en coordonnées polaires
private double ro;
private double tetha;
```

```
return ro;
```

```
return ro * Math.cos(theta);
```

```
return ro * Math.sin(theta);
```



Code utilisant la classe **Point**

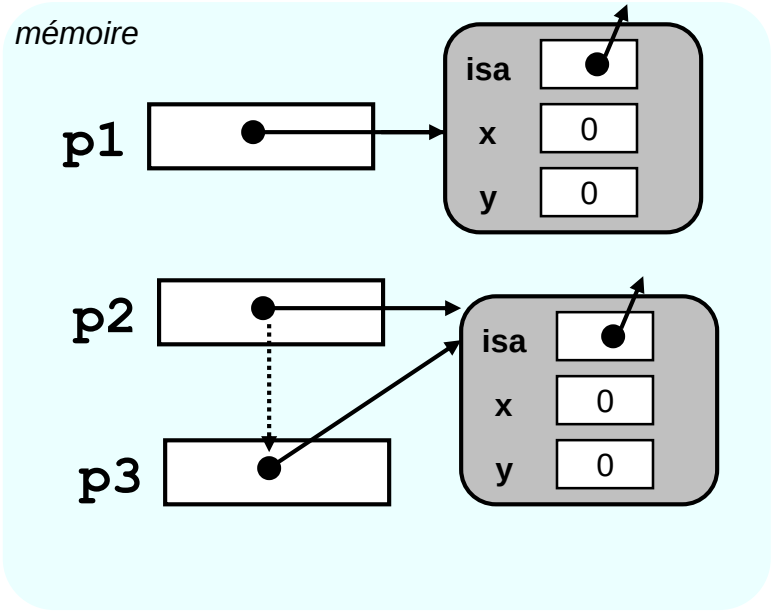
```
Point p1 = new Point();
...
p1.translater(x1,y1);
double d = p1.distance();
...
```

Modification de l'implémentation sans impact sur le code utilisant la classe si la partie publique (l'interface de la classe) demeure inchangée

Références et égalité d'objets

référence

```
Point p1;
p1 = new Point();
Point p2 = new Point();
Point p3 = p2;
```



égalité de références

```
p2 == p3        --> true
p1 == p2        --> false
```

égalité d ’objets
il faut passer par une méthode de la classe `Point`

```
p1.egale(p2)    --> true
```

```
public boolean egale(Point p) {
  return (this.x == p.x) && (this.y == p.y);
}
```

on reviendra sur les méthodes d'égalité d'objets dans le cours sur l'héritage

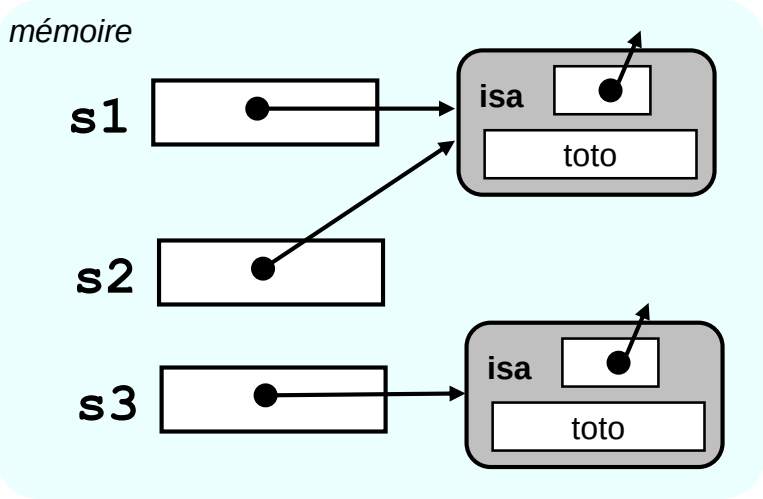
Références et égalité d'objets

```
String s1 = "toto";
String s2  = "toto";

System.out.println("s1 == s2 : " + (s1 == s2));

Scanner sc = new Scanner(System.in);
System.out.print("entrez une valeur : ");
String s3 = sc.nextLine();

System.out.println("s1 == s3 : " + (s1 == s3));
System.out.println("s1.equals(s3) : " + s1.equals(s3));
```

```
s1 == s2 : true

entrez une valeur :
toto

s1 == s3 : false
s1.equals(s3) : true
```

Pourquoi ?

Les `String` sont des objets !



`==` égalité de références

`equals` égalité de valeur d'objets