Java
Classes et Objets

Préparé par Larbi Hassouni

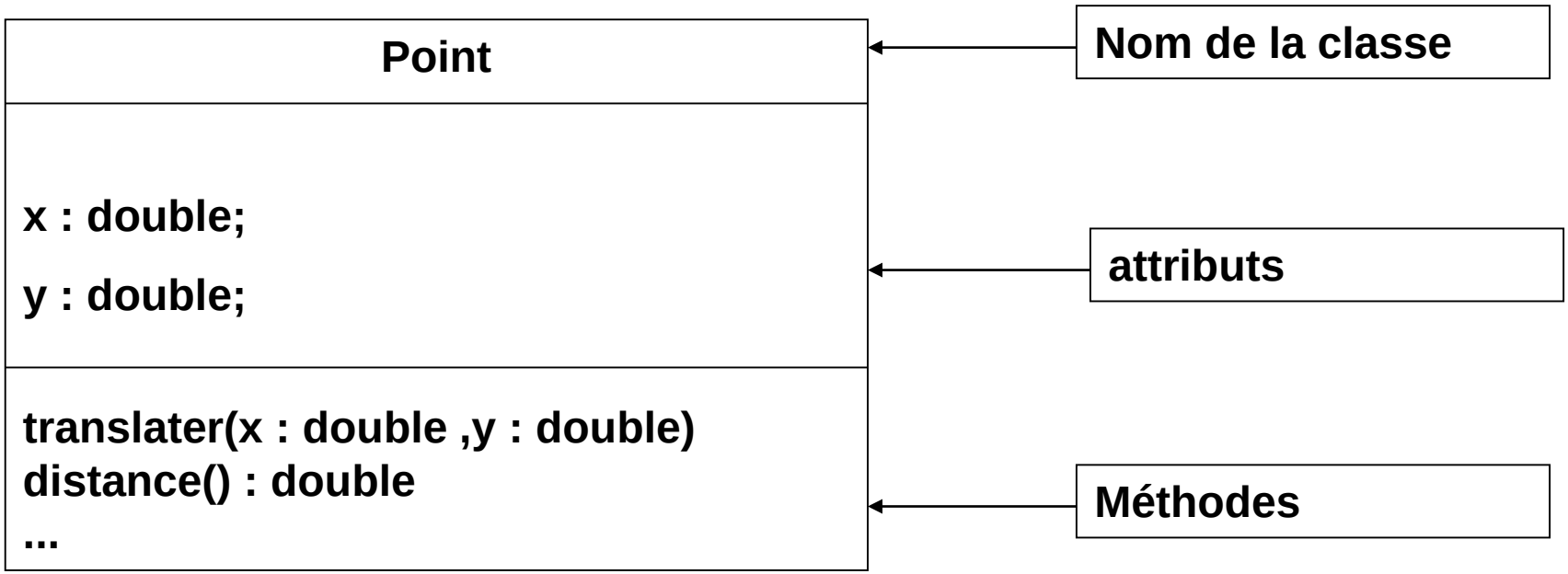
JAVA : Classes et Objets
(1ère partie)

- **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**.*
  - *procédures : que l'on nomme **méthodes***
- Une **classe** est un **modèle** de définition pour des objets
  - *ayant même structure (même ensemble d'attributs),*
  - *ayant même comportement (mêmes opérations, méthodes),*
  - *ayant une sémantique commune.*
- Les **objets** sont des représentations **dynamiques** (instanciation), « vivantes » 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*

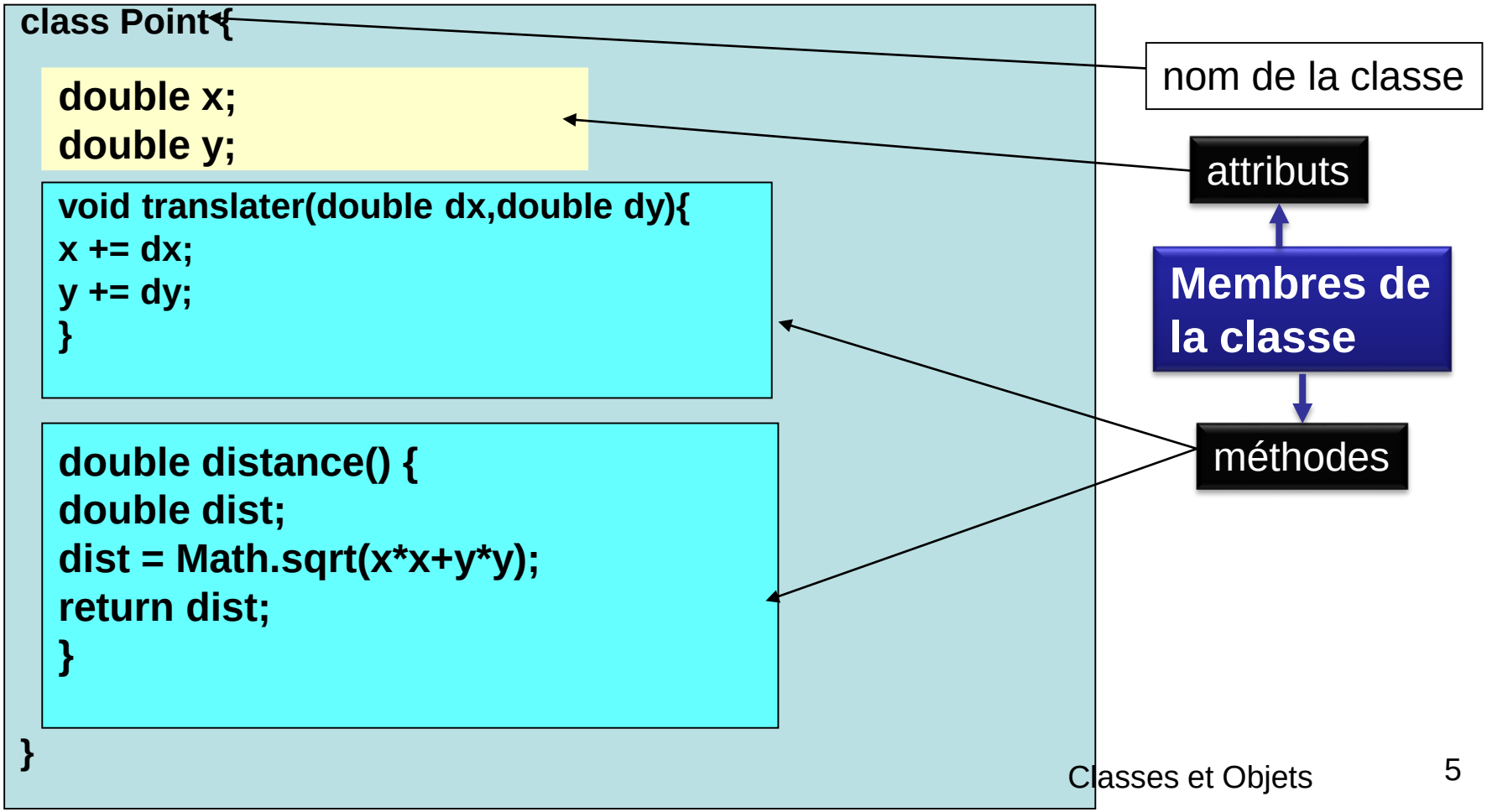
Classe : Notation UML



Classe : Syntaxe java

fichier Point.java
Attention :Le nom du fichier doit être identique au nom de la classe

```
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;
    }
}
```



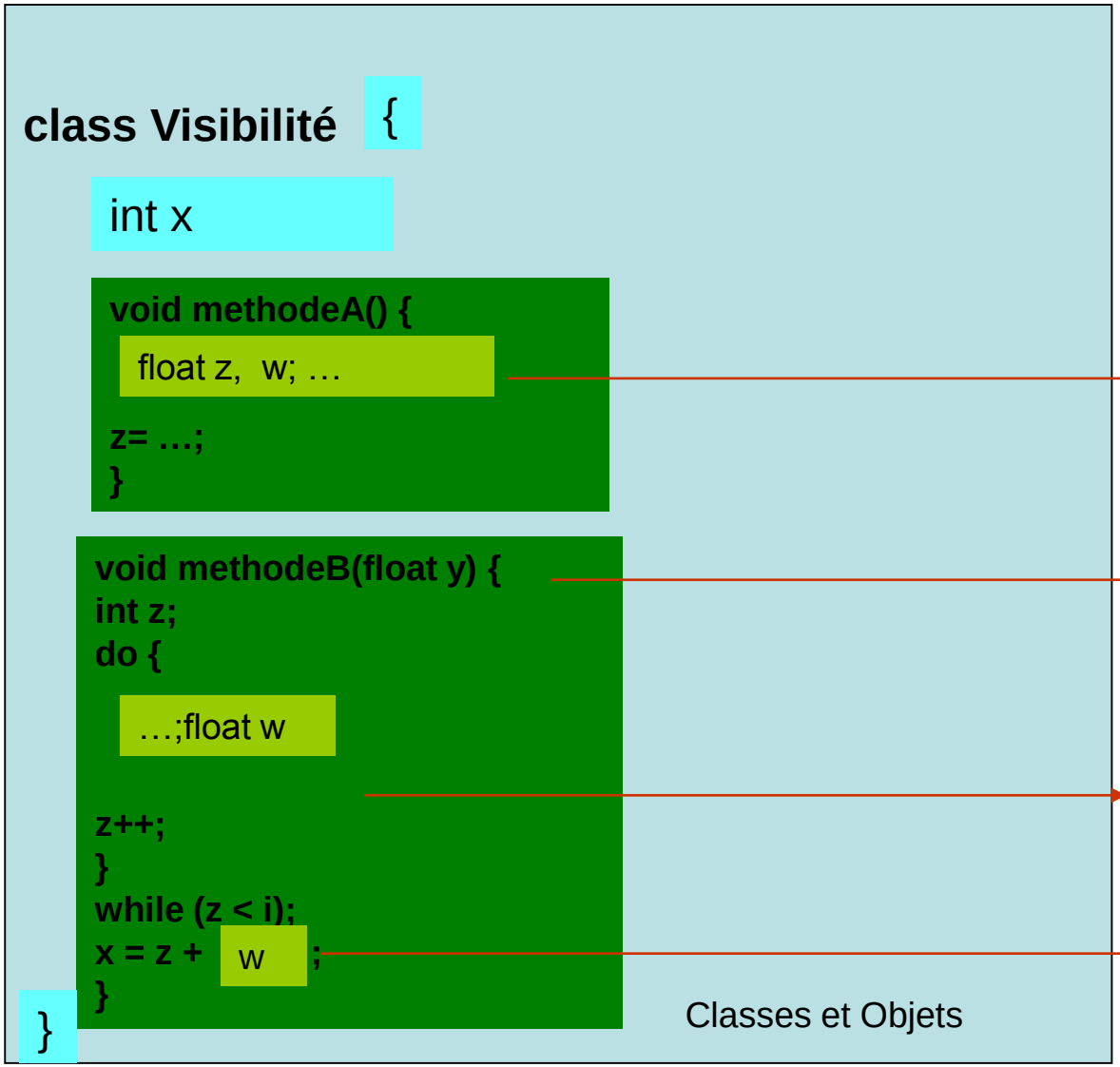
Classe : Syntaxe java
Visibilité des attributs

```
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 » au module que constitue la classe : ils sont accessibles dans Toutes les méthodes de la classe soit par leur simple nom soit en étant qualifiées par le mot this

Syntaxe java:Visibilité des variables

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



x:int z:float w:float

x:int y:float z:int

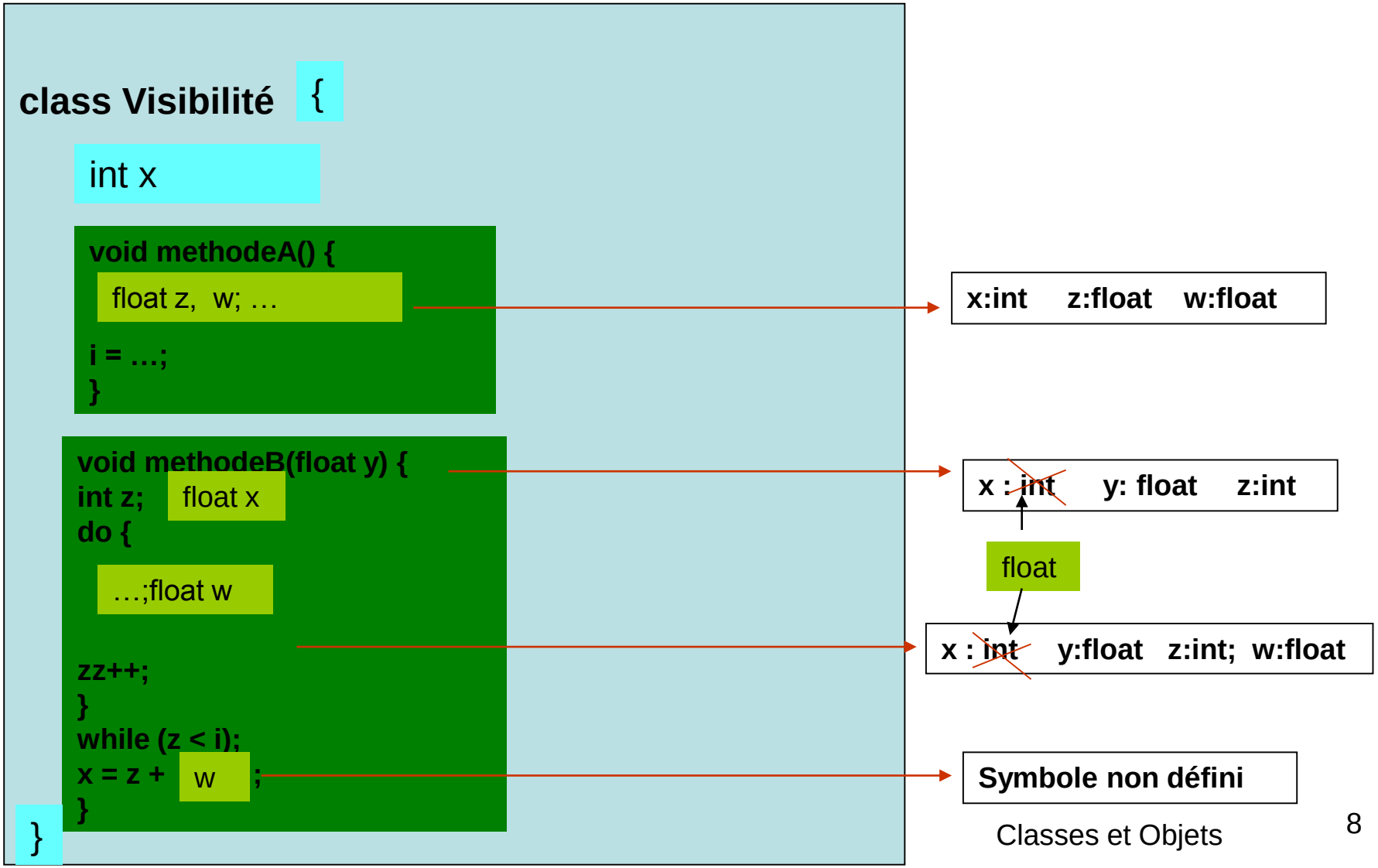
Variable peut être définie n'importe où dans un bloc

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

Symbole non défini

Syntaxe java:Visibilité des variables

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

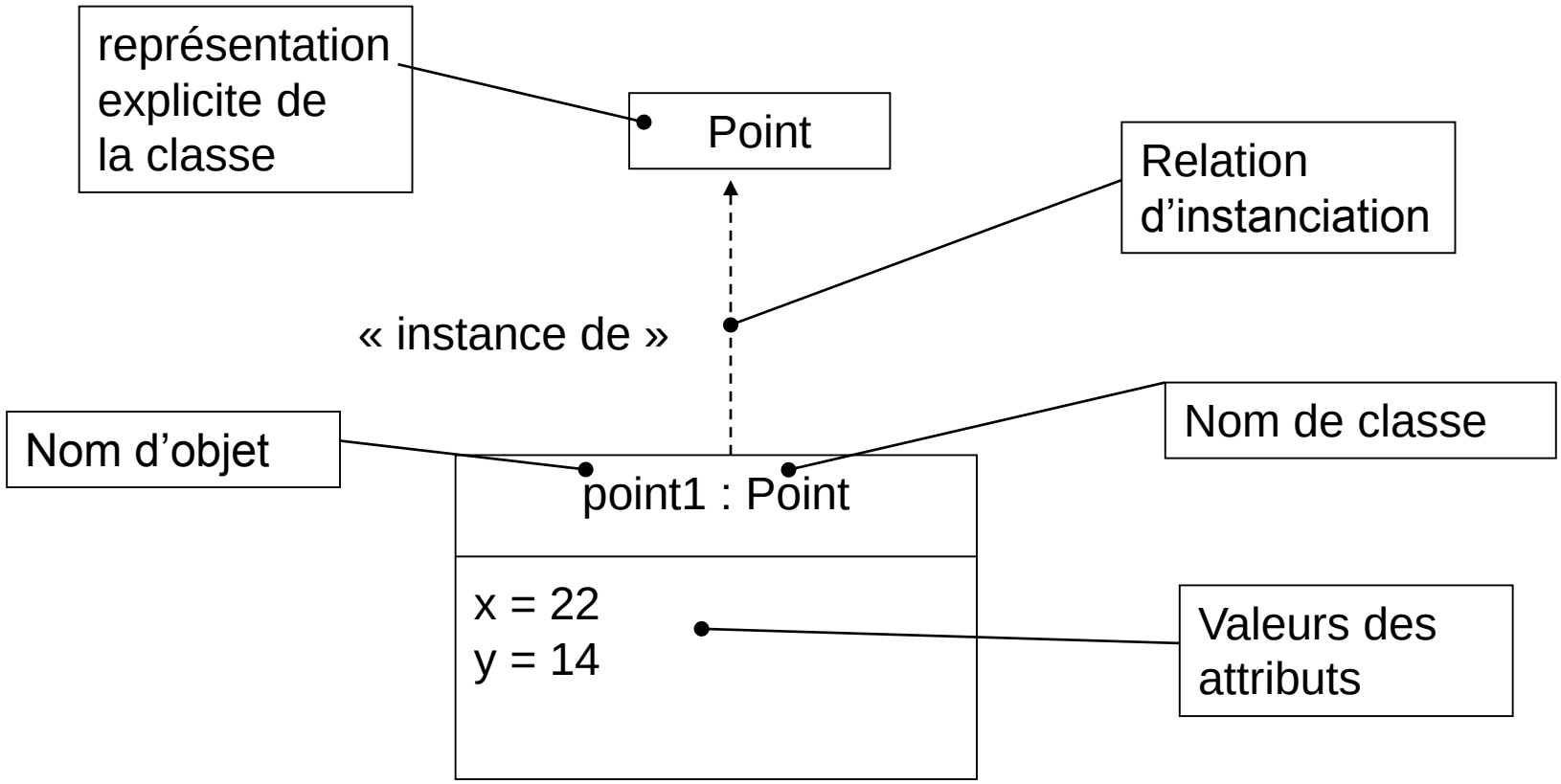


Objets

- 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*

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

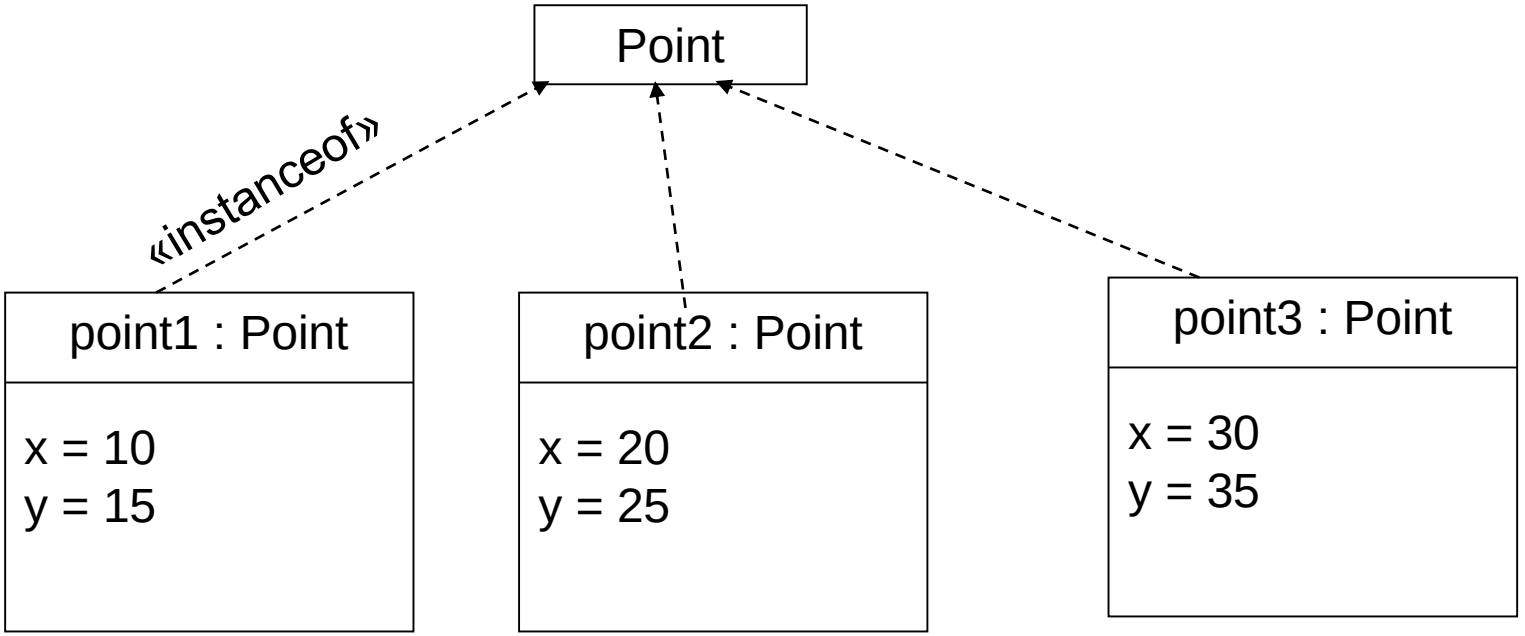
Objets : Notation UML



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

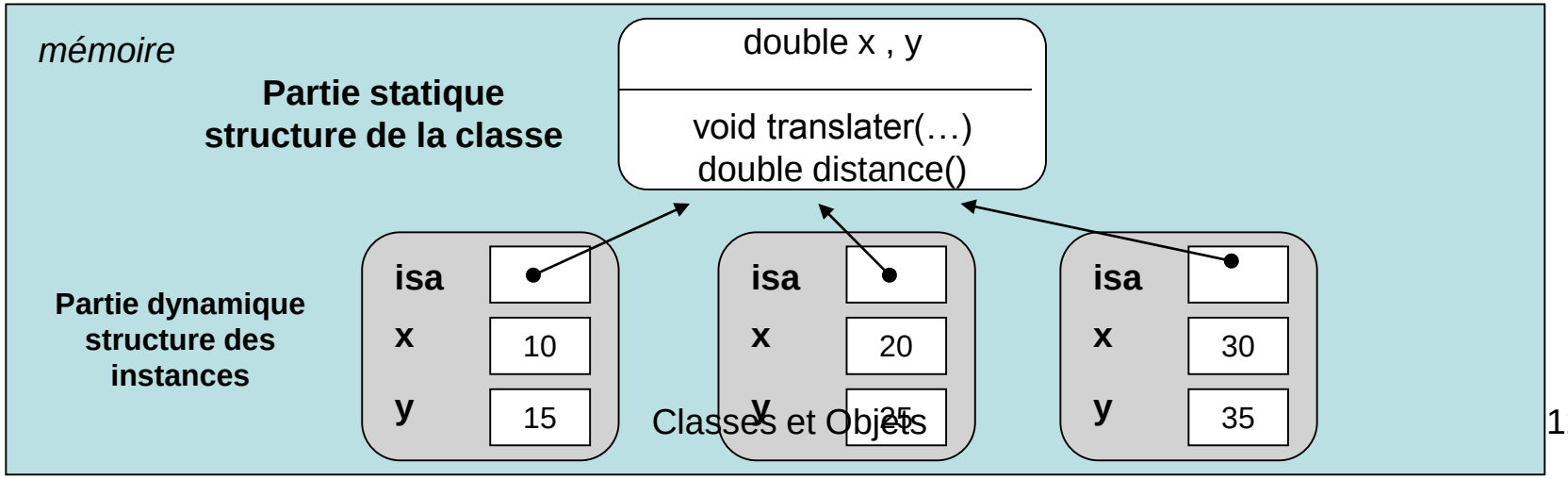
Objets

- Chaque objet point instance de la classe ***Point*** possédera son propre ***x*** et son propre ***y***



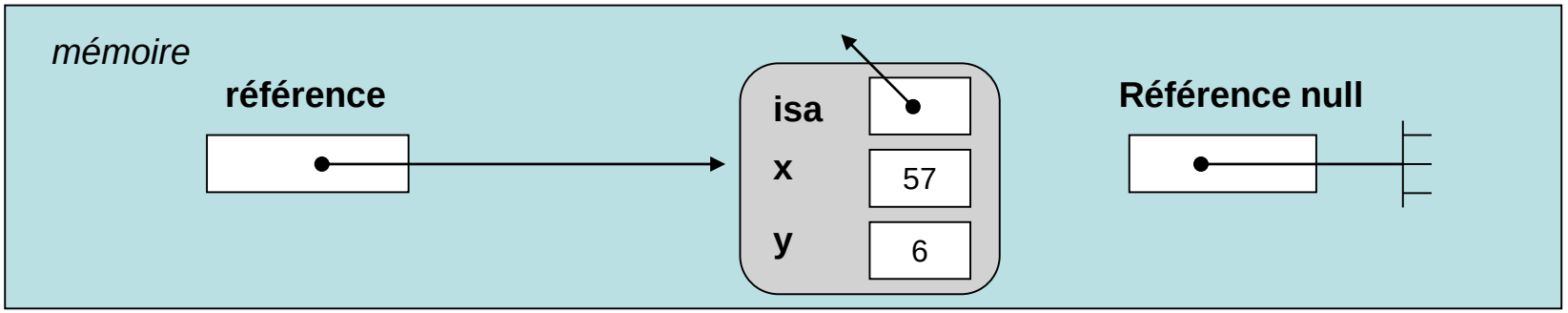
Objets : Structure des classes et objets en mémoire

- objet constitué d'une partie "**Statique**" et d'une partie "**Dynamique**"
  - *Partie statique :*
    - *ne varie pas d'une instance de classe à une autre*
    - *un seul exemplaire pour l'ensemble des instances d'une classe*
    - *constituée des méthodes de la classe*
    - *permet d'activer l'objet*
  - *Partie dynamique :*
    - *varie d'une instance de classe à une autre*
    - *varie durant la vie d'un objet*
    - *constituée d'un exemplaire de chaque attribut de la classe.*



Référence

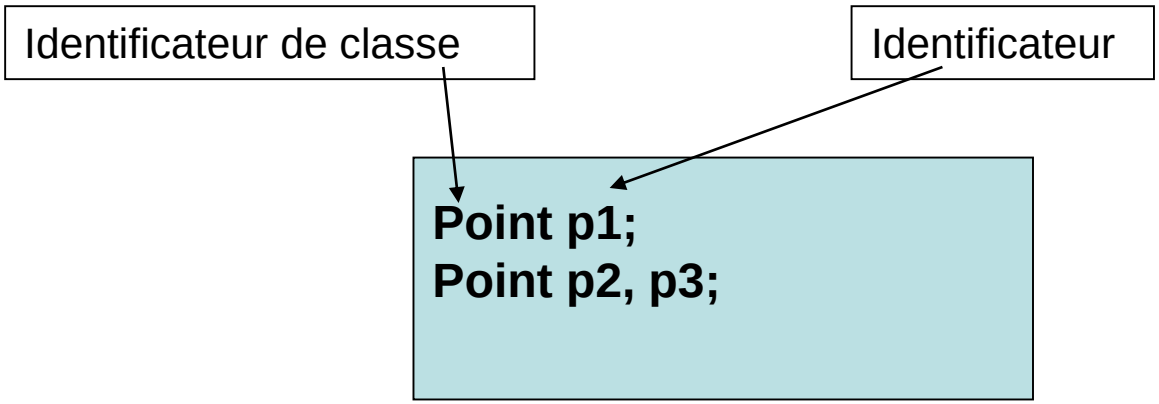
- Pour désigner des objets d'une même classe (attributs ou variables dans le corps d ’une méthode) on utilise des variables d’un type particulier : les **références**
- Une référence contient l’**adresse** d’un objet
  - *pointeur vers la structure de données correspondant aux attributs (variables d’instance)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

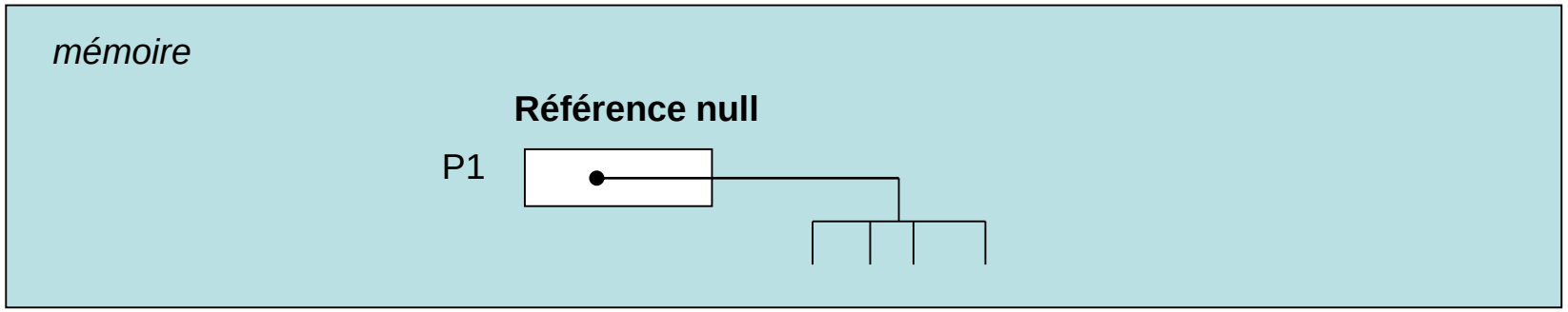
Référence : Déclaration en java

- Déclarations de références



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

Point p1; ⟺ **Point p1 = null;**



Références : Différents des pointeurs du C

- ***Comme un pointeur*** une référence contient une adresse (l'adresse de l'objet référencé)

- Mais ***à la différence des pointeurs*** les seules opérations autorisées sur les références est
  1. l'**affectation** d 'une référence de même type
  2. La **comparaison** avec l'opérateur égalité (==) de deux variables références de même type

```
Point p1;
…
Point p2;
p2 = p1;
if(p1==p2) ……
```

```
p1++;
…
p2 += p1 +3;
```

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 par défaut des attributs en mémoire*

  *2 : **appel de méthodes particulières**, les **constructeurs**, définies dans la classe ayant pour principal object initialiser les attributs pour donner un état initial à l'objet*

  *3 : renvoi d'une **référence sur l'objet** maintenant créé et initialisé.Cette référence est stockée dans une variable référence.*

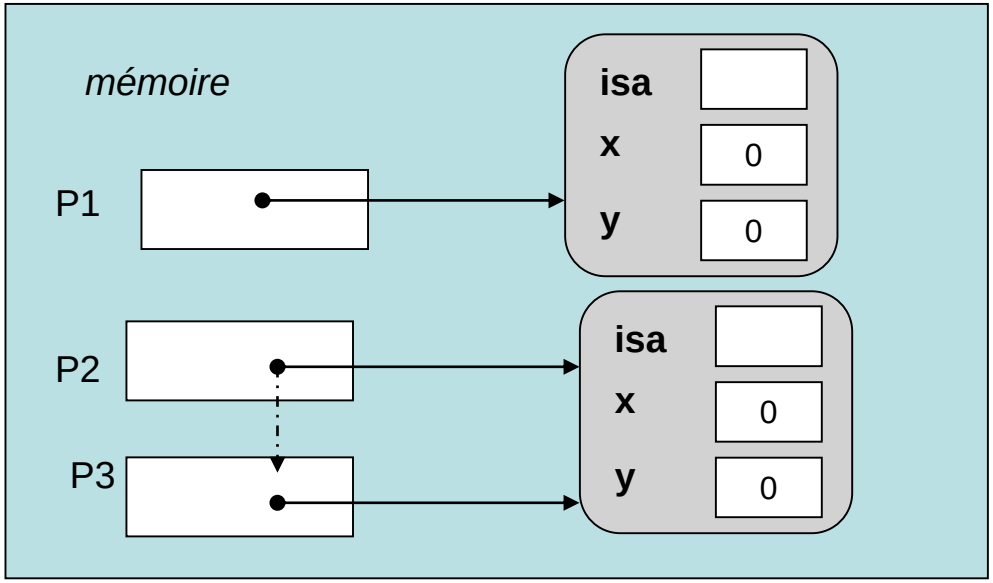
Création d'objets : Instanciation

- Pour créer un objet on fait appel à l'opérateur new suivi du constructeur de la classe: **new *constructeur(liste de paramètres)***

- les constructeurs ont le même nom que la classe

- il existe un constructeur par défaut:
  - *sans paramètres*
  - *n'effectue aucune initialisation. Il conserve les initialisations par défaut.*
  - ***inexistant si un autre constructeur existe***

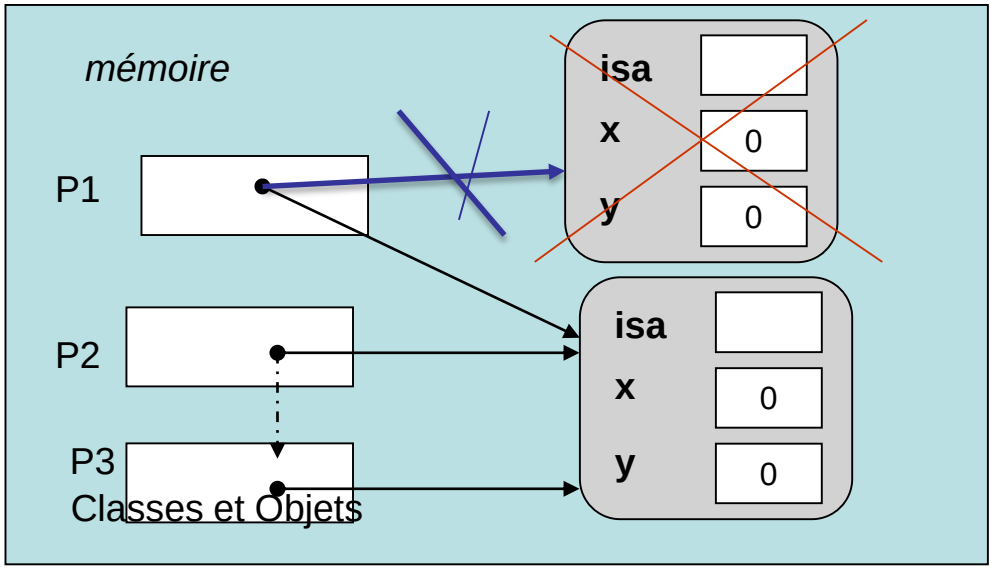
```
Point p1;

p1 = new Point();

Point p2 = new Point();

Point p3 = p2;
```



Création d'objets
Gestion de la mémoire

- 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 de la destruction (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;
```

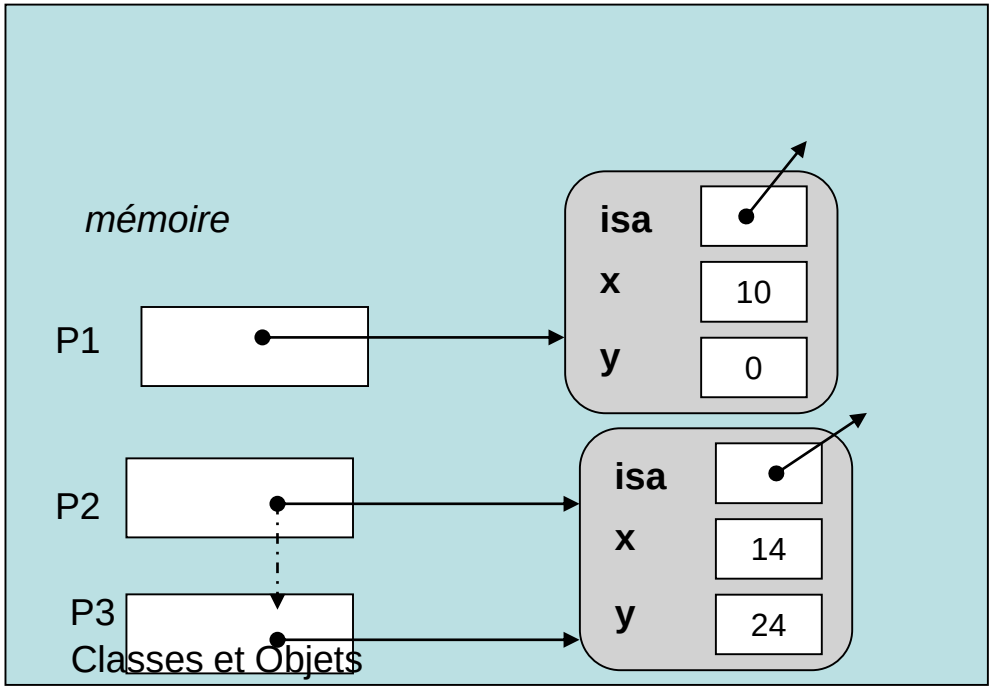


Accès aux attributs d'un objet

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

nomDeObjet.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 message

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

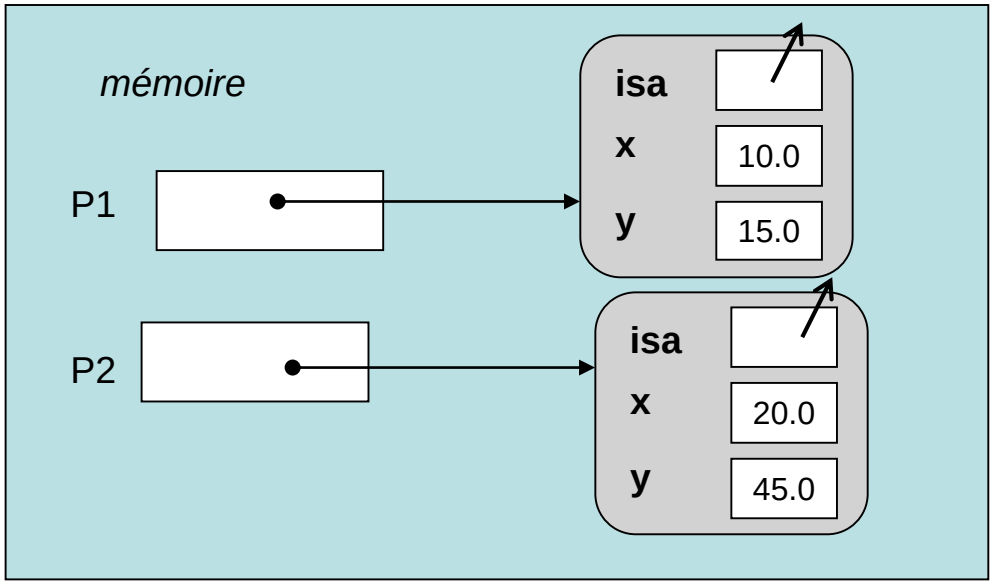
- un message est composé de trois parties
  - *une **référence** permettant de désigner l'objet à qui le message est envoyé*
  - *le **nom de la méthode** à exécuter (cette méthode doit bien entendu être définie dans la classe de l'objet)*
  - *les éventuels **paramètres** de la méthode*

- 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 message : Exemple

- syntaxe :
  - **nomDeObjet.nomDeMethode(<paramètres effectifs>)**

```
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(10.0,15.0);
p2.translater(2*p1.x,3.0 * p1.y);
System.out.println("distance de p1 à origine «+ p1.distance());
```

Envoi d'un message à l'objet p1

Envoi d'un message à l'objet p2

Envoi de message
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 : savoir si un Point est plus proche de l'origine qu'un autre Point.*

Envoi de message
Paramètres des méthodes (suite)

Ajout d'une méthode à la classe Point

```
/**
* Test 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 plusProcheOrigineQue(Point p) {
        if (this.distance() < p.distance())
                return true;
        else
                return false;
}
```

Utilisation

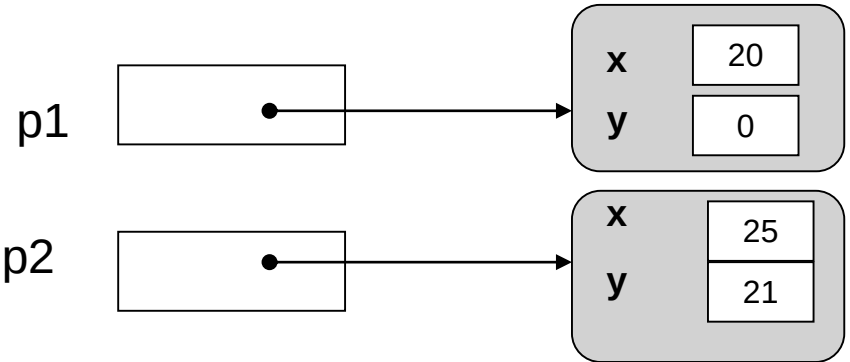
```
Point p1 = new Point();
Point p2 = new Point();
P1.translater(10,15);
 p2.translater(20,5);
if (p1.plusProcheORigineQue(p2))
        System.out.println("p1 est plus proche de l'origine que p2");
else
        System.out.println("p2 est plus proche de l'origine que p1");
```

Envoi de message
Paramètres des méthodes (suite)

- 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.*

```
Class Point {
...
void foo(double x, Point p)
{

p.translater(10,10);
x = x + 10;
p = new Point();
p.translater(10,10);

}
}
```

```
p1 = new Point();
p2 = new Point();
p2.x = 15; p2.y = 11;
p1.x = 20;
p1.foo(p1.x, p2);
System.out.println(« p1.x " +p1. x);
System.out.println("p2.x " + p2.x);
System.out.println("p2.y " + p2.y);
```



L'objet courant : Désigné par le mot clé this

- Dans un message l'accent est mis sur l'objet (et nom pas sur l'appel de fonction)
  - *en JAVA (et de manière plus générale en POO) on écrit :*
    d1= p1.distance(); d2=p2.distance();
- l'objet qui reçoit un message est implicitement passé comme argument à la méthode invoquée
- cet argument implicite défini par le mot clé **this**
  - *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 envoi de message

Pour invoquer, dans le code d'une classe, l'une des méthodes qu'elle définit (récursivité possible)

```
class Point {
double x;
double y;
// constructeurs
Point(double dx, double dy){
...
}
// méthodes
boolean plusProcheDeOrigineQue(Point p){

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


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

- **L'objet qui reçoit le message se renvoie à lui-même un autre message**
- **this n'est pas indispensable**
- **l'ordre de définition des méthodes n'a pas d 'importance**

L'objet courant
This et variables d'instance

Implicitement, quand Un attribut est utilisé dans le corps d'une Méthode, il s'agit de l'attribut de l'objet courant

***this est, quelque fois utilisé,* essentiellement pour lever les ambiguités**

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

Encapsulation

- Il est possible d'accéder directement aux variables d'un objet

- L'accès direct aux variables est non recommandé car :
  - contraire au principe d'encapsulation
  - *les données d'un objet doivent être privées (c'est à dire protégées)*
  - *Elles ne doivent être accessible en lecture et écriture qu'au travers de méthodes prévues à cet effet.*

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

- Le niveau de visibilité d'un membre (attribut, méthode, ou constructeur) peut être défini en précédant sa déclaration d'un modificateur (**private**, **public**, **protected**, **-**)

Encapsulation
Visibilité des membres d'une classe

	public	**private**	**protected**	**Pas de modificateur**
classe	Peut être utilisée dans n'importe quelle autre classe de n'importe quel package	N/A	N/A	Peut être utilisée uniquement par les classes de son package
attribut	Accessible directement depuis le code de n'importe quelle autre classe	Accessible uniquement dans le code de la classe qui le définit	Accessible dans les classes de même package et dans les sous classes	Accessible dans les classes de même package
méthode	Peut être invoquée à partir du code de n'importe quelle autre classe	Peut être invoquée uniquement dans la classe qui la définit	Peut être invoquée dans les classes de même package et dans les sous classes	Peut être invoquée dans les classes de même package

Encapsulation
Méthodes d'accés aux attributs privés (accesseurs : accessors)

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

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

```
Point p1 = new Point();
p1.x = 10;   ⟹   p1.setX(10);
p1.y = 10;       p1.setY(10);
Point p2 = new Point();
p2.x = p1.x;  ⟹  p2.setX(p1.getX());
p2.y = p1.x + p1.y;  ⟹
          p2.setY(p1.getX()+p1.getY());
```

- *pour modifier un attribut privé il faut passer par une méthode de type procédure (appelée accesseur ou modificateur)*

- *pour accéder à la valeur d'un attribut privé il faut passer par une méthode de type fonction (appelé accesseur)*

Encapsulation : usage des 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 p.distance() < this.distance();
   }
   ...
}
```

```
public class X {
    Point p=new Point(…);
     …
     p.distance()
      …
}
```

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

• la méthode privée distance a été définie pour un usage interne à la classe. Elle est invoquée par d'autres méthodes de sa classe

Encapsulation : Intérêts = Robustesse du code

- **L'Accès aux données ne se fait qu'au travers des méthodes définies par le concepteur de la classe.**

Un objet ne peut être utilisé que selon la manière prévue lors de la conception de sa classe. Ce qui élimine le risque d'utilisation incohérente : **Robustesse du code**

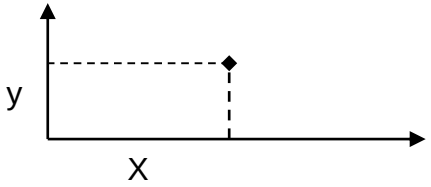
```
// représentation d'un point de l'écran
public class Pixel {
   // représentation en coordonnées cartésiennes
   private int x; // 0 <= x < 1024
   private int y; // 0 <= y < 780
   public int getX() {
      return x;
   }
   public void translater(int dx,int dy) {
      if ( ((x + dx) < 1024) && ((x + dx) >= 0) )
         x = x +dx;
      if ( ((y + dy) < 780) && ((y + dy) >= 0) )
         y = y + dy;
      }
    ...
} // Pixel
```

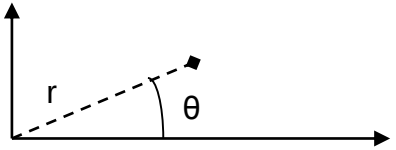
Code utilisant la classe Pixel

```
Pixel p1 = new Pixel();

p1.translater(100,100);

//le message ci-dessus  modifiera les
//coordonnées de p1: x = 100, y = 100.

p1.translater(1000,-300);

//Le message ci-dessus ne modifiera pas
//les coordonnées de p1, puisque x
//deviendrait =1100>1024 et y = -200<0
...
```

Encapsulation : Intérêts = Evolution du code

❑ **Masquer l'implémentation** ⇒ **Facilite d'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 void translater(double dx,double dy) {
      …..
      }
    ...
} // Point
```

```
// représentation d'un point de l'écran
public class Pixel {
   // représentation en coordonnées polaires
   private double r
   private double θ;
  public double distance() {
      return r;
   }
 public void translater(double dr,double dθ) {
      ……
      }
   ...
} // Point
```

Code utilisant la classe Point

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

La modification de l'implémentation n'a pas d'impact sur le code utilisant la classe si la partie publique (l'interface de la classe) demeure inchangée

JAVA : Classes et Objets 2ème partie

- Constructeurs
- Surcharge des méthodes
- Variables de classe
- Méthodes de classe
- Constantes
- La fonction **main()**
- Initialiseur statique
- Initialiseur d'instance
- Finalisation

Constructeurs

❑ **Constructeurs d'une classe :**

- Un constructeur est une *méthode particulière utilisée pour la création des objets de cette classe*
- Le nom d'un constructeur *est identique au nom de la classe*

rôle d'un constructeur

effectuer certaines initialisations nécessaires pour le nouvel objet créé

Chaque classe JAVA possède au moins un constructeur

si une classe ne définit pas explicitement de constructeur, un constructeur par défaut sans arguments et qui n'effectue aucune initialisation particulière est invoqué.

Constructeurs : Définition explicite

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

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

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

Déclaration explicite d ’un constructeur

- Le constructeur par défaut est masqué
- nom du constructeur identique à celui de la classe
- pas de **type de retour** ni mot void dans la signature
- retourne implicitement une instance de la classe (this)
- pas d ’instruction return dans le constructeur

Création d ’objet

~~**new Point()**~~

new Point(15,14)

Constructeurs multiples

- Possibilité de définir plusieurs constructeurs dans une même classe
  ⇨ *possibilité d'initialiser un objet de plusieurs manières différentes*

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

une classe peut définir un nombre quelconque de constructeurs

•chaque constructeur possède le même nom (le nom de la classe)

• le compilateur distingue les constructeurs en fonction :
  -du nombre
  -du type
  -de la position des arguments

• on dit que les constructeurs peuvent être **surchargés** (**overloaded**)

Surcharge des méthodes

- La **surcharge** (overloading) n’est pas limitée aux constructeurs, elle est possible pour n'importe quelle méthode

- Il est possible de définir des méthodes possédant le même nom mais dont les paramètres diffèrent

- lorsque qu'une méthode surchargée est invoquée *le compilateur sélectionne automatiquement la méthode dont le nombre et le type des arguments correspondent au nombre et au type des paramètres passés dans l'appel de la méthode*

- des méthodes surchargées peuvent avoir des types de retour différents mais à condition qu'elles aient des arguments différents

Surcharge des méthodes : Exemple

```
public class Point {
// attributs
private double x;
private double y;
// constructeurs
public Point(double x, double y){
   this.x = x;
   this.y = y
}
// méthodes
public double distance() {
return Math.sqrt(x*x+y*y);
}
 public double distance(Point p){
 return Math.sqrt((x - p.x)*(x - p.x)
 + (y - p.y) * ( y - p.y));
 }
…
}
```

```
Point p1=new Point(10,10);
Point p2=new Point(15,14);

p1.distance();

p1.distance(p2);
```

Constructeurs

Appel d'un constructeur par un autre constructeur

- dans les classes définissant plusieurs constructeurs, un constructeur peut invoquer un autre constructeur de cette classe en utilisant le mot **this**.

- l'appel **this(...)**
  - *fait référence au constructeur de la classe dont les arguments correspondent à ceux spécifié après le mot this*

  - ***ne peut être utilisé que comme première instruction dans le corps d'un constructeur, il ne peut pas être invoqué après d'autres instructions***

  - *(on comprendra mieux cela lorsque l'on parlera de l'héritage et de l'invocation automatique des constructeurs de la superclasse)*

Constructeurs

Appel d'un constructeur par un autre constructeur

```
public class Point {
private double x;
private double y;
// constructeurs
private Point(double x, double y)
{
      this.x = x;
      this.y = y;
}
public Point(Point p)
{
       this(p.x, p.y);
}
public Point()
{
       this(0.0, 0.0)
}
...
}
```

Constructeur général

Possibilité de définir des constructeurs privés

Intérêt :
- **Factorisation du code.**
- **Un constructeur général invoqué par des constructeurs particuliers.**

Variables de classe

```
public class Point {
   private double x;       // abscisse du point
   private double y;       // ordonnée du point

   // Définition des fonctions abscisse() et ordonnées()
   ...
/* Compare 2 points cartésiens
* @return true si les points sont égaux à 1.0e-5 près
*/
   public boolean egale(Point p) {
     double dx= x - p.abscisse();
     double dy= y - p.ordonnee();
     if(dx<0)
        dx = -dx;
     if(dy<0
        dy= - dy;
      return (dx < 1.0e-5 && dy < 1.0e-5);
   }
}
```

Modifier la classe Point
afin de pouvoir modifier la valeur de la constante de précision

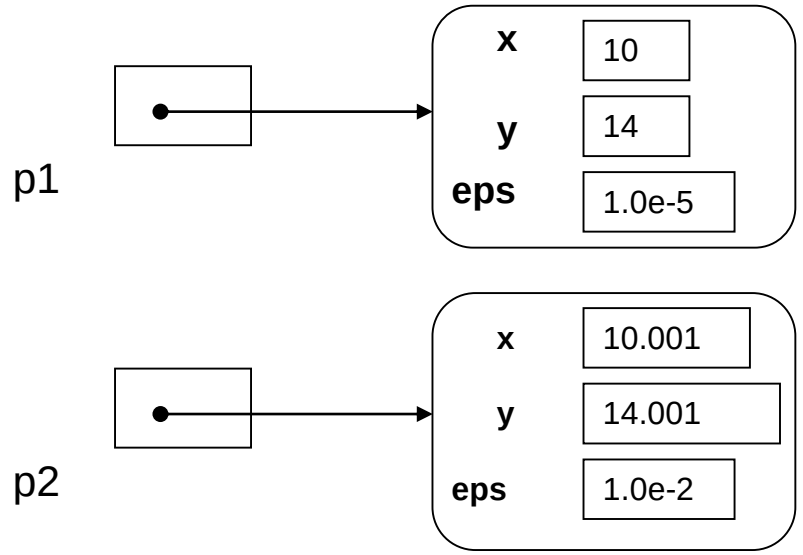
Variables de classe

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

// imprécision pour tests d'égalité
private double eps = 1.0e-5;
//modifie la valeur de l' imprécision
public void setEps(double eps) {
this.eps = eps;
}
//restitue valeur imprécision
public double getEps() {
return eps;
}
……..
// Compare 2 points cartésien
public boolean egale(Point p) {
double dx= x - p.abscisse();
double dy= y - p.ordonnee();
if(dx<0)  dx = -dx;
if(dy<0) dy= - dy;
return (dx <  eps && dy < eps );
}
…..
}
```

1ère solution :
Ajouter une variable (un attribut) eps à la classe avec une méthode accesseur et une méthode « modificateur »

Quels sont les problèmes liés à cette solution ?

Variables de classe

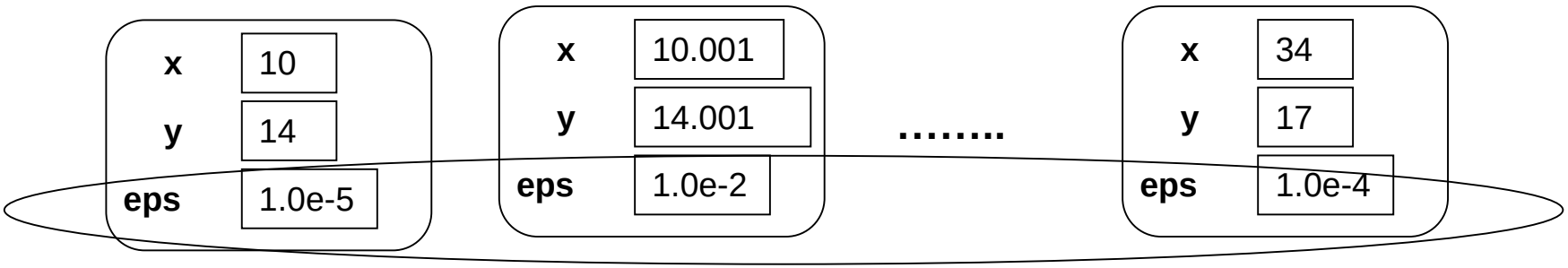


Point p1 = new Point(10,14);
Point p2 = new Point(10.001,14.001);
P1.setEps(1 E-5);
p2.setEps(10E-2);
System.out.println(p1.egale(p2)); -=> false on utilise la précision de p1 (=0.00001)

System.out.println(p2.egale(p1)); => true on utilise la précision de p2(= 0.01)

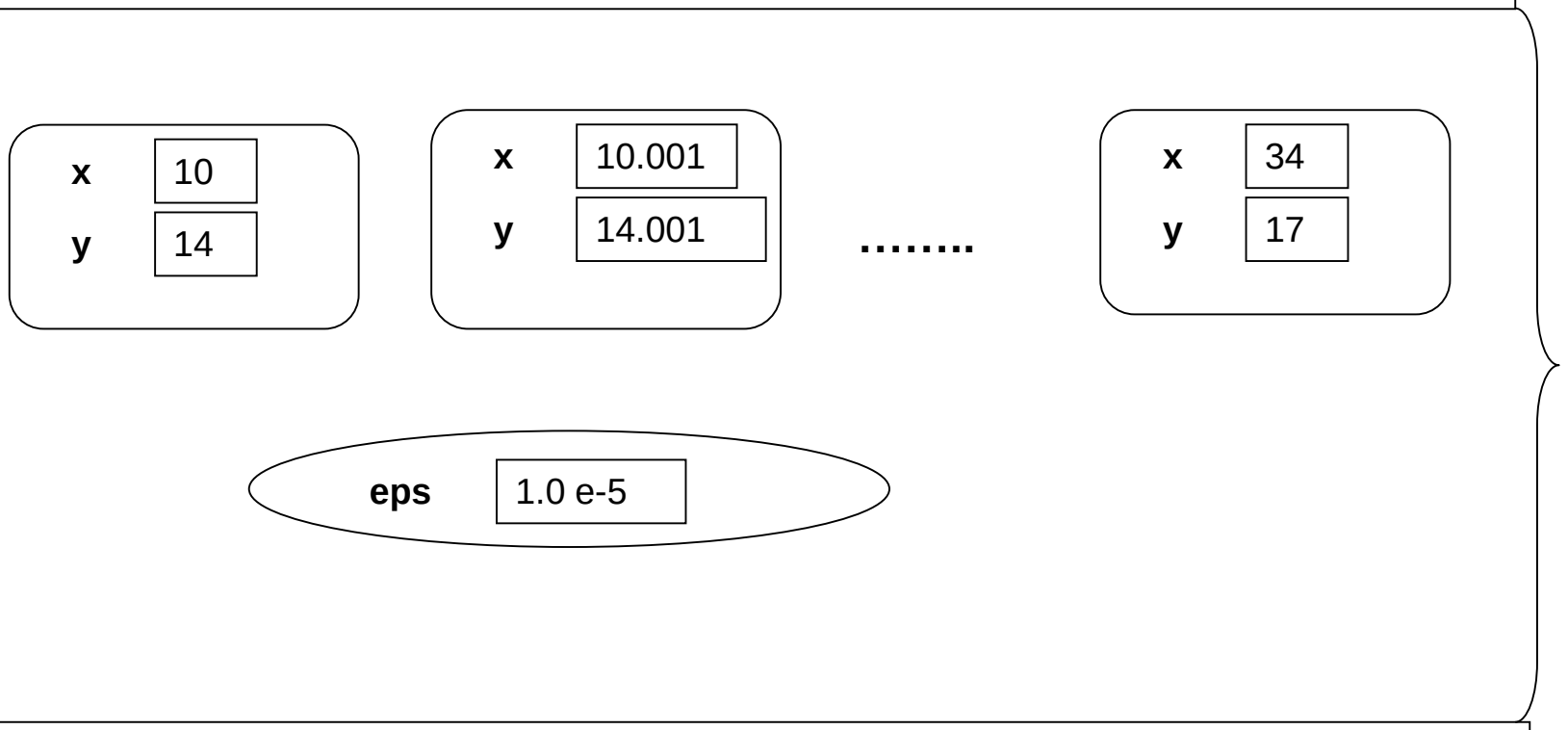
Chaque instance possède sa propre valeur de précision. **egale** n'est plus garantie comme étant symétrique

P1.egale(p2) ≠ p2.egale(p1)



Variables de classe

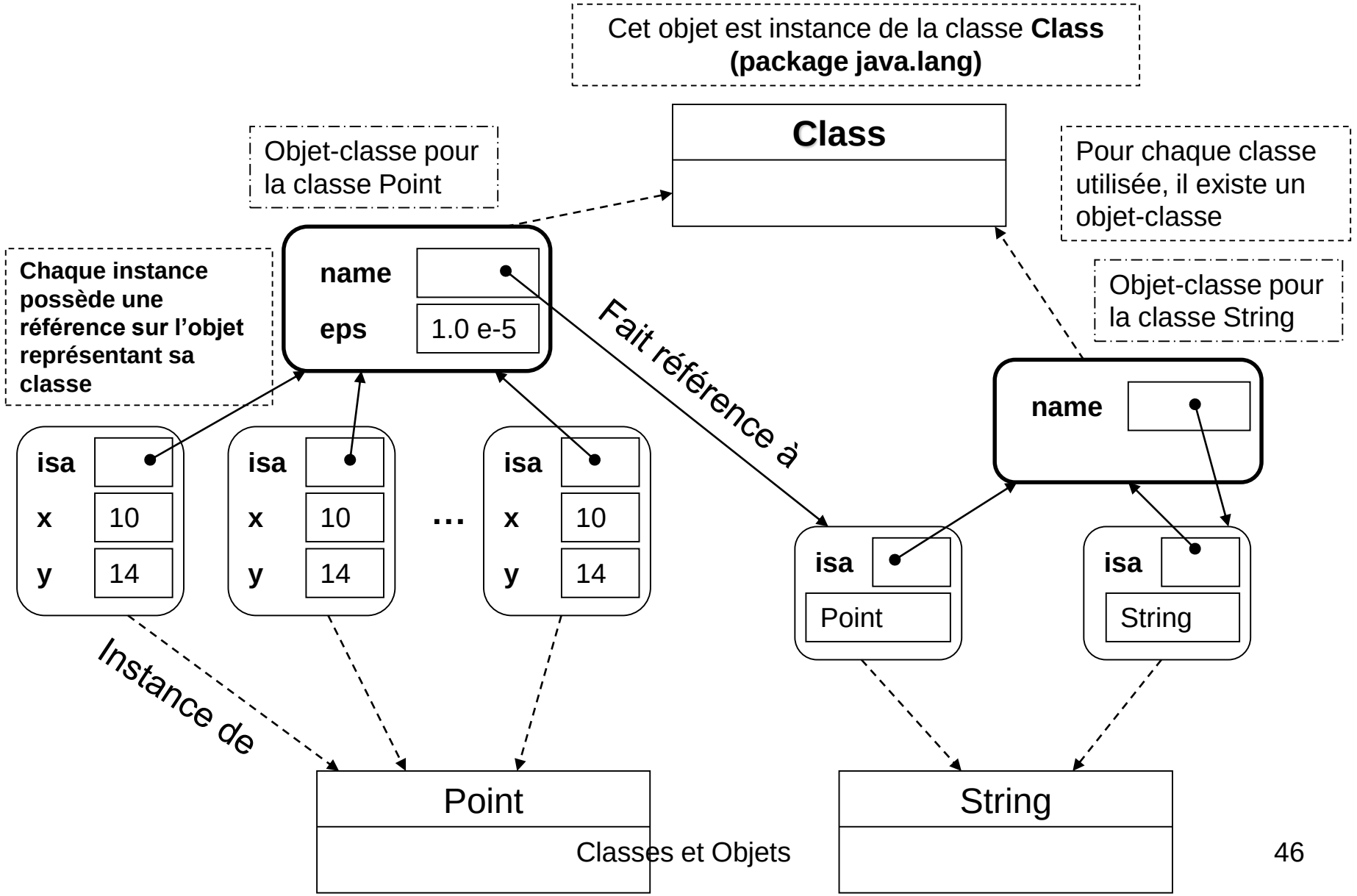
Cette information ne concerne pas une instance particulière mais l'ensemble du domaine des **Point**



Comment représenter cet ensemble ?

En Java, rien ne peut être défini en dehors d'un objet. pas de variables globales

Variables de classe



API de la classe Class

public String **getName**()

Returns the name of the entity (class, interface, array class, primitive type, or void) represented by this Class object, as a String.

public Class **getSuperclass**()

Returns the Class representing the superclass of the entity (class, interface, primitive type or void) represented by this Class

public Object **newInstance**() throws InstantiationException, IllegalAccessException

Creates a new instance of the class represented by this Class object.

…

Les membres d'une classes sont eux-mêmes représentés par des objets dont les classes sont définies dans java.lang.reflect. (Introspection des objets)

public Field **getField**(String name) throws NoSuchFieldException, SecurityException

Returns a Field object that reflects the specified public member field of the class or interfacerepresented by this Class object.

public Method[] **getMethods**() throws SecurityException

Returns an array containing Method objects reflecting all the public *member* methods of the class or interface represented by this Class object, including those declared by the class or interface and those inherited from superclasses and superinterfaces.

Exemple de programme utilisant la class **Class**

Extrait du livre :

Au cœur de Java 2

Notions fondamentales

De Cay S.Horstmann & Gary Cornell

(Page : 225)

```
import java.util.*;
import java.lang.reflect.*;

public class ReflectionTest
{
   public static void main(String[] args)
   {
      // read class name from command line args or user input
      String name;
      if (args.length > 0)
         name = args[0];
      else
      {
         Scanner in = new Scanner(System.in);
         System.out.println("Enter class name (e.g. java.util.Date): ");
         name = in.next();
      }

      try
      {
         // print class name and superclass name (if != Object)
         Class cl = Class.forName(name);
         Class supercl = cl.getSuperclass();
         System.out.print("class " + name);
         if (supercl != null && supercl != Object.class)
            System.out.print(" extends " + supercl.getName());
```

Exemple de programme utilisant la class **Class (suite 1)**

```
System.out.print("\n{\n");
        printConstructors(cl);
        System.out.println();
        printMethods(cl);
        System.out.println();
        printFields(cl);
        System.out.println("}");
      }
      catch(ClassNotFoundException e) { e.printStackTrace(); }
      System.exit(0);
   }

   /**
      Prints all constructors of a class
      @param cl a class
    */
   public static void printConstructors(Class cl)
   {
      Constructor[] constructors = cl.getDeclaredConstructors();

      for (Constructor c : constructors)
      {
         String name = c.getName();
         System.out.print("   " + Modifier.toString(c.getModifiers()));
         System.out.print(" " + name + "(");
```

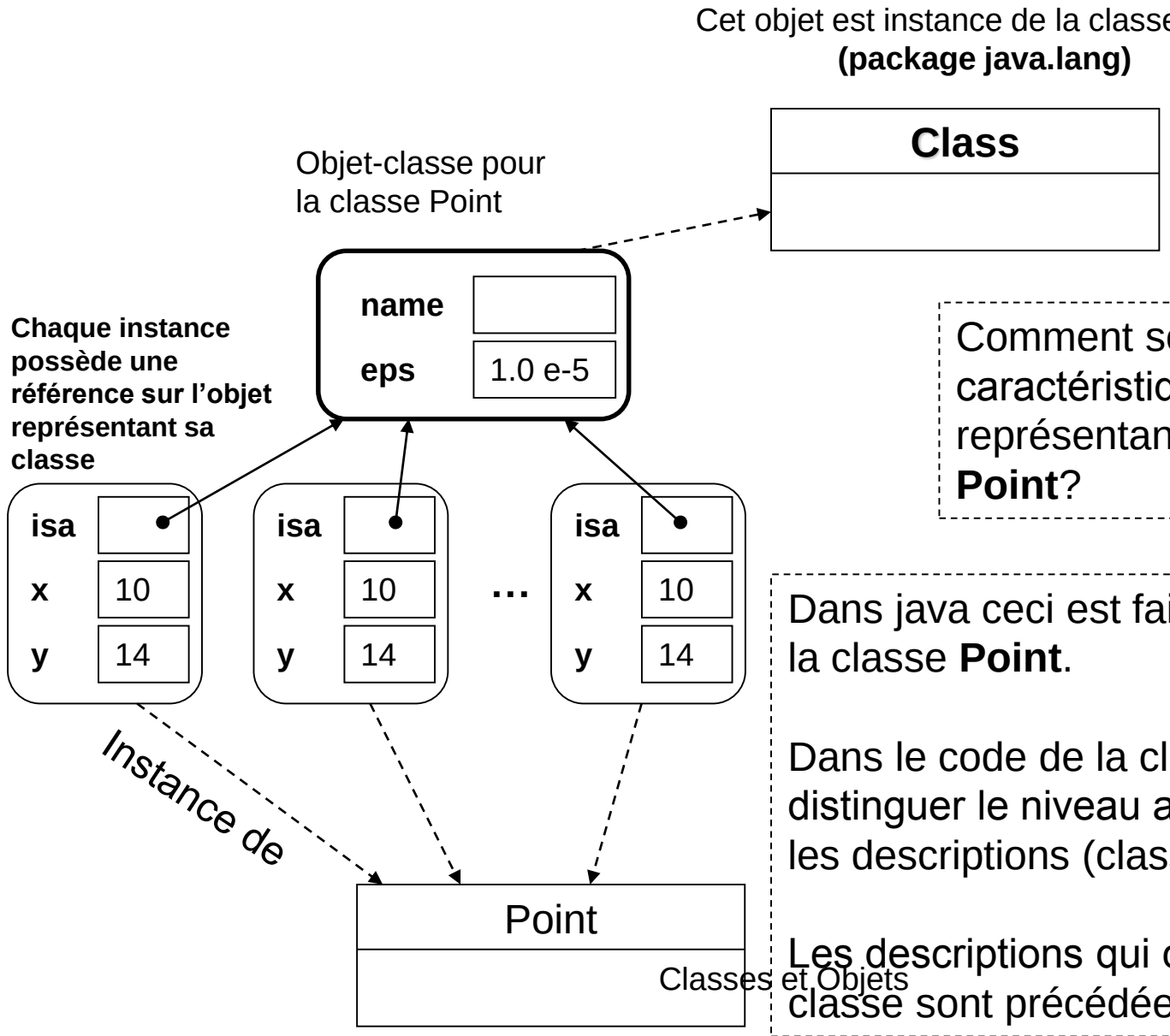
Exemple de programme utilisant la class **Class (suite 2)**

```
// print parameter types
        Class[] paramTypes = c.getParameterTypes();
        for (int j = 0; j < paramTypes.length; j++)
        {
           if (j > 0) System.out.print(", ");
           System.out.print(paramTypes[j].getName());
        }
        System.out.println(");");
     }
  }

  /**
     Prints all methods of a class
     @param cl a class
   */
  public static void printMethods(Class cl)
  {
     Method[] methods = cl.getDeclaredMethods();

     for (Method m : methods)
     {
        Class retType = m.getReturnType();
        String name = m.getName();
// print modifiers, return type and method name
        System.out.print("   " + Modifier.toString(m.getModifiers()));
        System.out.print(" " + retType.getName() + " " + name + "(");
```

Exemple de programme utilisant la class **Class (suite 3)**

```
// print parameter types
        Class[] paramTypes = m.getParameterTypes();
        for (int j = 0; j < paramTypes.length; j++)
        {
           if (j > 0) System.out.print(", ");
           System.out.print(paramTypes[j].getName());
        }
        System.out.println(");");
     }
  }

  /**
     Prints all fields of a class
     @param cl a class
   */
  public static void printFields(Class cl)
  {
     Field[] fields = cl.getDeclaredFields();

     for (Field f : fields)
     {
        Class type = f.getType();
        String name = f.getName();
        System.out.print("   " + Modifier.toString(f.getModifiers()));
        System.out.println(" " + type.getName() + " " + name + ";");
     }
  }
}
```

Variables de classe



Comment sont décrites les caractéristiques de l'objet représentant la classe **Point**?

Dans java ceci est fait au sein du code de la classe **Point**.

Dans le code de la classe **Point** il faut distinguer le niveau auquel s'appliquent les descriptions (classe ou aux instances)

Les descriptions qui concernent l'objet-classe sont précédées du mot clé **static**

Variables de classe : Déclaration

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

// imprécision pour tests d'égalité
private static double eps = 1.0e-5;
public void setEps(double eps) {
this.eps = eps;
}

public double getEps() {
return eps;
}
……
// Compare 2 points cartésiens
public boolean equals(Point p) {
double dx= x - p.abscisse();
double dy= y - p.ordonnee();
if(dx<0)
dx = -dx;
if(dy<0
dy= - dy;
return (dx < eps && dy < eps);
}
……..
}
```

Variable d'instance

Variable de classe déclaration précédée du mot clé **static**

Dans le corps de la classe les variables de classe sont utilisées comme les autres variables… ou presque

Accès aux Variables de classe

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

// imprécision pour tests d'égalité
private static double eps = 1.0e-5;
public void setEps(double eps) {
this.eps = eps;
}

public double getEps() {
return eps;
}
……
// Compare 2 points cartésiens
public boolean equals(Point p) {
double dx= x - p.abscisse();
double dy= y - p.ordonnee();
if(dx<0)
dx = -dx;
if(dy<0
dy= - dy;
return (dx < eps && dy < eps);
}
……..
}
```

Warning :The static field eps should be accessed in a static way

Accéder à eps en dehors de la classe Point

il serait nécessaire de posséder une référence sur une instance de **Point** pour lui envoyer un message **getEps** ou **setEps**

```
Point p = new Point (…);
…
p.setEps(1.e-8);
```

eps n'est pas un attribut de l'objet Point (this) mais un attribut de l'objet représentant la classe Point

Comment faire pour désigner l'objet classe Point ?

Accès aux Variables de classe

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

// imprécision pour tests d'égalité
private static double eps = 1.0e-5;
public void setEps(double eps) {
this.eps = eps;
}
```

Warning :The static field eps should be accessed in a static way

```
public double getEps() {
return eps;
}
……
// Compare 2 points cartésiens
public boolean equals(Point p) {
double dx= x - p.abscisse();
double dy= y - p.ordonnee();
if(dx<0)
dx = -dx;
if(dy<0
dy= - dy;
return (dx < eps && dy < eps);
}
……..
}
```

de la même manière que les variables d'instance sont accédées via les noms (références) des instances de la classe, les variables de classe sont accédées au travers du nom de la classe

NomDeClasse.nomDeVariable

le nom de la classe est l'identificateur de la référence créée par défaut pour désigner l'objet-classe

this.x **Variable d'instance**

Point.eps **Variable de classe**

Méthodes de classe

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

// imprécision pour tests d'égalité
private static double eps = 1.0e-5;

Public static void setEps(double eps) {
Point.eps = eps;
}

public static double getEps() {
return eps;
}
……
// Compare 2 points cartésiens
public boolean equals(Point p) {
double dx= x - p.abscisse();
double dy= y - p.ordonnee();
if(dx<0)
dx = -dx;
if(dy<0
dy= - dy;
return (dx < eps && dy < eps);
}
}
```

Les méthodes **setEps** et **getEps** ne doivent plus être associées aux instances de la classe **Point** mais à l'objet-classe **Point**

Déclaration de méthodes de classe (méthodes statiques)

Attention : à l'intérieur du corps d'une méthode statique il n'est possible d'accéder qu'aux membres statiques de la classe

Appel d'une méthode de classe : Envoi d'un message à l'objetcclasse

Point.setEps(1.e-8);

Variables de classe : Constantes

Des **constantes nommées** peuvent être définies par des variables de classe dont la valeur ne peut être modifiée

```
public class Pixel {

  public static final int MAX_X = 1024;
  public static final int MAX_Y = 768;

// variables d'instance
private int x;
private int y;
...
// constructeurs
// crée un Pixel de coordonnées x,y
public Pixel()
{
...
}
...
}
```

Déclarations de variables de classe

Le modifieur **final** est utilisé pour indiquer que la valeur d'une variable (ici de classe) ne peut jamais être changée

Membres statiques

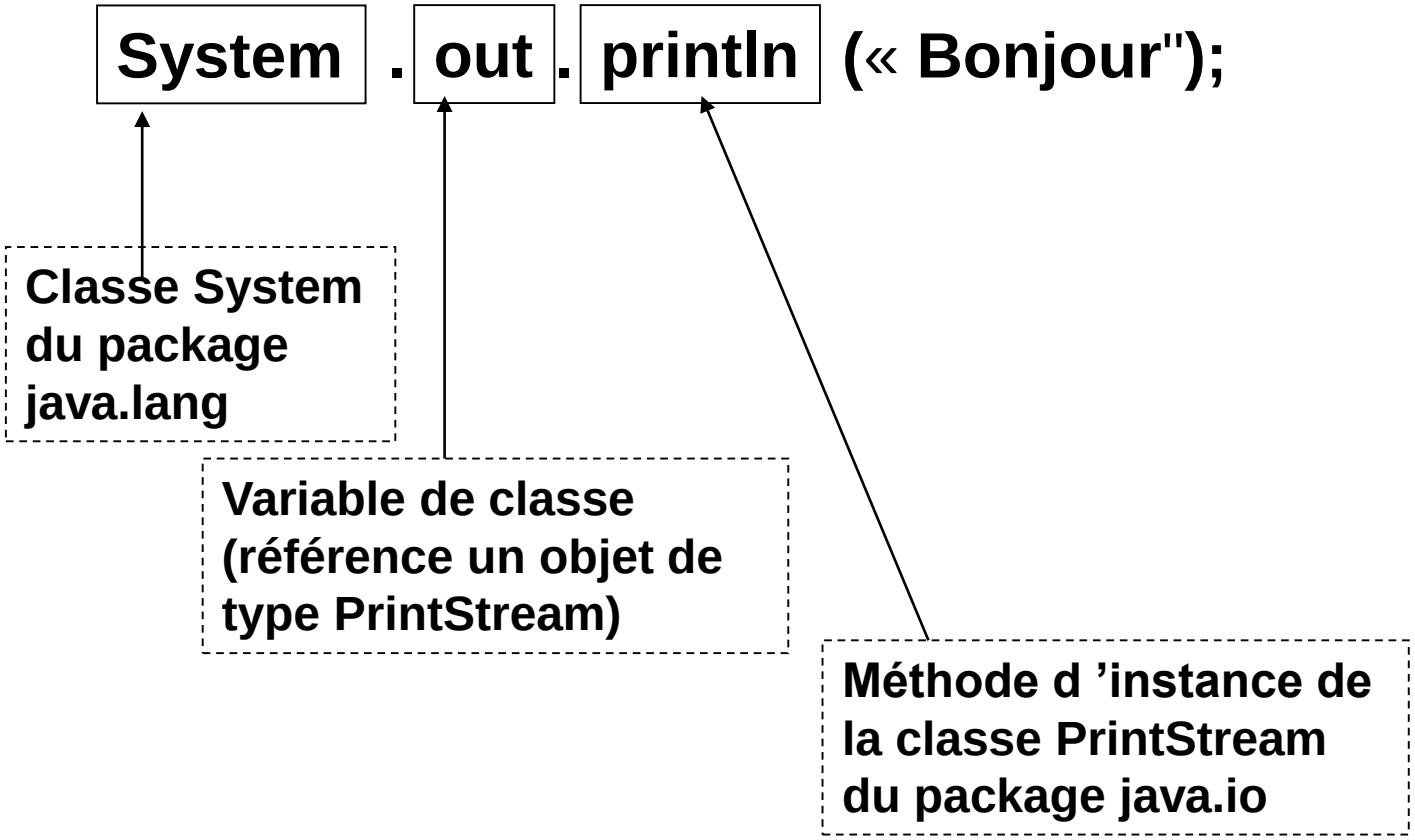
- Ce sont des membres dont la déclaration est précédée du modificateur ***static***
  - ***variables de classe*** *: définies et existent indépendamment des instances*
  - ***méthodes de classe*** *: dont l'invocation peut être effectuée sans passer par l'envoi d'un message à l'une des instances de la classe.*

- accès aux membres statiques
  - *directement par leur nom dans le code de la classe où ils sont définis,*
  - *en les préfixant du nom de la classe en dehors du code de la classe*

    - **NomDeLaClasse.nomDeLaVariable**
    - **NomDeLaClasse.nomDeLaMéthode(liste de paramètres)**
    - *n'est pas conditionné par l'existence d'instances de la classe,*

      Math.PI **Math.cos(x)** **Math.toRadians(90) ...**

Signification de System.out.println



public static void main(String[] args)

- Le point d ’entrée pour l’exécution d ’une application Java est la méthode statique **main** de la classe spécifiée à la machine virtuelle

- la syntaxe de cette méthode est :

public static void main(String[] args)

- **String[] args**

args : tableau d’objets String (chaînes de caractères) contenant les arguments de la ligne de commande

```
public class TestArgs {
public static void main(String[] args) {
System.out.println("nombre d ’arguments : " + args.length);
for (int i =0; i < args.length; i++)
System.out.println(" argument " + i + " = " + args[i]);
}
}
```

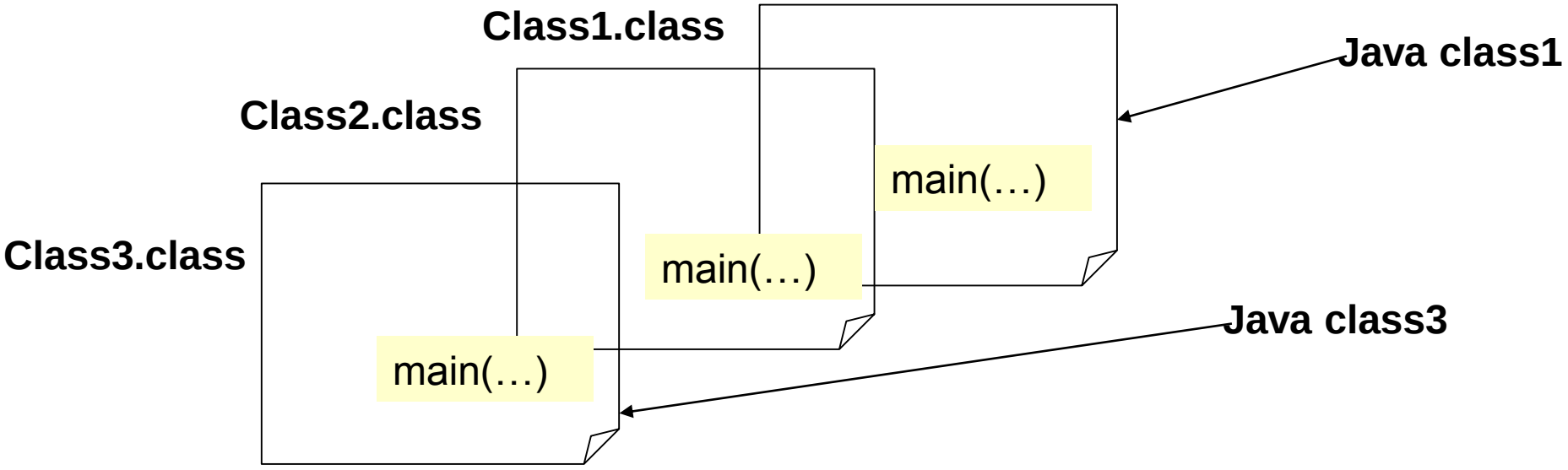
Java TestArgs arg1 20 35.0 →

```
nombre d ’arguments : 3
argument 0 : arg1
argument 1 : 20
argument 2 : 35.0
```

public static void main(String[] args)

- Les différentes classes d'une même application peuvent éventuellement **chacune** contenir leur propre méthode main.

- Au moment de l'exécution pas d'hésitation quant à la méthode **main()** à exécuter
  - *C'est celle de la classe indiquée à la JVM*



- Interêt
  - *possibilité de définir de manière indépendante un test unitaire pour chacune des classes d'un système*

Initialisation des variables : à la déclaration

- les variables d'instance et de classe peuvent avoir des "**initialiseurs**" associés à leur déclaration

modifieurs type nomDeVariable **= *expression*;**

```
private double x = 10;
private double y = x + 2;
private double z = Math.cos(Math.PI / 2);
private static int nbPoints = 0;
```

- variables de classe initialisées la première fois que la classe est chargée.
- variables d'instance initialisées lorsqu'un objet est créé.
- les initialisations ont lieu dans l'ordre des déclarations.

```
public class TestInit {
private double y = x + 1;
private double x = 14.0;
...
}
```

```
TestInit.java [4:1] illegal forward reference
private double y = x + 1;
                   ^

1 error
Errors compiling TestInit.
```

Initialisation des variables : initialiseurs statiques

- si les initialisations nécessaires pour les variables de classe ne peuvent être faites directement avec les initialiseurs (expression) JAVA permet d'écrire une méthode (algorithme) d'initialisation pour celles-ci : **l'initialiseur statique**

xs un nombre tiré au hasard entre 0 et 10
ys somme de n nombres tirés au hasard, n étant la partie entière de xs
zs somme de xs et ys

```
private static double xs = 10 * Math.random();
private static double ys ;
private static double zs = xs + ys;
static {
int n = (int) xs;
ys = 0;
for (int i = 0; i < n; i++)
ys = ys + Math.random();
}
```

- **Déclaration d'un initialiseur statique**
  - **static {** *bloc de code* **}**
  - *méthode*
    - *sans paramètres*
    - *sans valeur de retour*
    - *sans nom*

- **Invocation**
  - ***automatique*** *et* ***une seule fois*** *lorsque la classe est chargée*
  - *dans l'ordre d'apparition dans le code de la classe*

Exemple d'utilisation des blocs d'initialisation statique et d'objets

```
/**@version 1.01 2004-02-19
   @author Cay Horstmann
*/
import java.util.*;
public class ConstructorTest
{
   public static void main(String[] args)
   {
      // fill the staff array with three Employee objects
      Employee[] staff = new Employee[3];
      staff[0] = new Employee("Harry", 40000);
      staff[1] = new Employee(60000);
      staff[2] = new Employee();
      // print out information about all Employee objects
      for (Employee e : staff)
         System.out.println("name=" + e.getName()
            + ",id=" + e.getId()
            + ",salary=" + e.getSalary());
   }
}
```

```
class Employee{
   // three overloaded constructors
   public Employee(String n, double s){
      name = n;
      salary = s;
      System.out.println("Execution du 1er constructeur");
    }
   public Employee(double s) {
      // calls the Employee(String, double) constructor
      this("Employee #" + nextId, s);
 System.out.println("Execution du 2eme constructeur");
}
   // the default constructor
   public Employee()   {
      // name initialized to ""--see below
      // salary not explicitly set--initialized to 0
      // id initialized in initialization block
System.out.println("Execution du constructeur par defaut");
}
    public String getName()   {  return name;   }
   public double getSalary()  {  return salary;   }
   public int getId()  {  return id;   }
```

```
   private static int nextId;

   private int id;
   private String name = ""; // instance field initialization
   private double salary;

   // static initialization block
   static
   {
      Random generator = new Random();
      // set nextId to a random number between 0 and 9999
      nextId = generator.nextInt(10000);
      System.out.println("Execution du bloc static********");
   }

   // object initialization block
   {
      id = nextId;
      nextId++;
System.out.println("Execution du bloc d'instance: Creation d'objet
Id:"+id);

   }
}
```

Destruction des objets : Rappels

- La libération de la mémoire alloué aux objets est automatique
  - *lorsqu'un objet n'est plus référencé le "ramasse miettes" ("garbage collector") récupère l'espace mémoire qui lui était réservé.*
- Le "ramasse miettes" est un processus (thread) qui s'exécute en tâche de fond avec une priorité faible
  - *s'exécute :*
    - *lorsqu'il n'y a pas d'autre activité (attente d'une entrée clavier ou d'un événement souris)*
    - *lorsque l'interpréteur JAVA n'a plus de mémoire disponible*
      - *seul moment où il s'exécute alors que d'autres activités avec une priorité plus forte sont en cours (et ralentit donc réellement le système)*
- peut être moins efficace que la gestion explicite de la mémoire , mais programmation beaucoup plus simple et sure.

Destruction des objets : Finalisation

- Le "ramasse miettes" gère automatiquement les ressources mémoire utilisées par les objets

- un objet peut parfois détenir d'autres ressources (descripteurs de fichiers, sockets....) qu'il faut libérer lorsque l'objet est détruit.

- Il faut prévoir une méthode dite de "**finalisation**" à cet effet pour
  - *fermer les fichiers ouverts, terminer les connexions réseau..., avant la destruction des objets*

- la méthode de "finalisation" :
  - *est une méthode d'instance*
  - *doit être appelée* **finalize()**
  - *ne possède pas de paramètres , n'a pas de type de retour (retourne void)*
  - *est invoquée juste avant que le "ramasse miette" ne récupère l'espace mémoire associé à l'objet*
- JAVA ne donne aucune garantie sur quand et dans quel ordre la récupération de la mémoire sera effectuée,
  - *impossible de faire des suppositions sur l'ordre dans lequel les méthodes de finalisation seront invoquées.*

Destruction des objets : Finalisation exemple

```
class Point {
private double x;
private double y;
public Point(double x, double y) {
this.x = x;
this.y = y;
}
public void translater(double dx, double dy) {
x += dx;
y += dy;
}
public String toString() {
return "Point[x:" + x + ", y:" + y + "]";
}
public void finalize() {
System.out.println("finalisation de " + this);
}
}
```

```
Point p1 = new Point(14,14);
Point p2 = new Point(10,10);
System.out.println(p1);
System.out.println(p2);
p1.translater(10,10);
p1 = null;
System.gc();
System.out.pritnln(p1);
System.out.pritnln(p2);
```

Appel explicite du Garbage Collector

```
Point[x:14.0, y:14.0]
Point[x:10.0, y:10.0]
finalisation de Point[x:24.0, y:24.0]
null
Point[x:10.0, y:10.0]
```