Classes et Objets

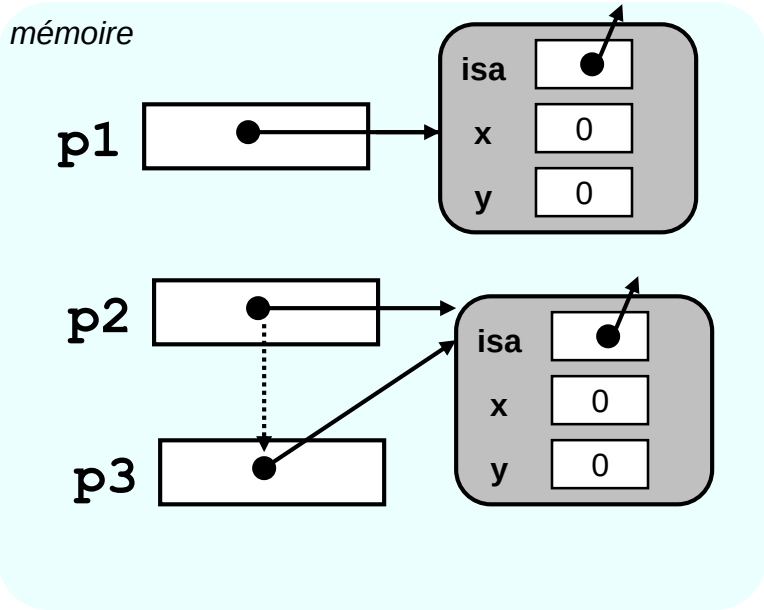
2ème partie

- Rappels
- Références et égalité d'objets
- Constructeurs
- Surcharge des méthodes
- Variables de classe
- Méthodes de classe
- Constantes
- Le `main()`
- Initialiseur statique
- Finalisation

Références et égalité d'objets

référence

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



égalité de références

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

égalité d 'objets

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

il faut passer par une méthode de la classe `Point`

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

Références et égalité d'objets

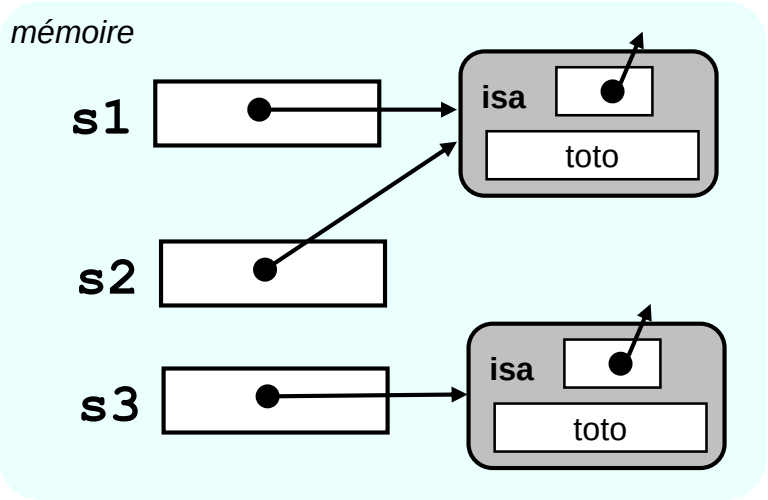
les String

```
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**

Constructeurs

le retour

- **Constructeurs** d'une classe :
  - *méthodes particulières pour la création d'objets de cette classe*
  - *méthodes dont le nom est identique au nom de la classe*

- rôle d'un constructeur
  - *effectuer certaines initialisations nécessaires pour le nouvel objet créé*

- toute 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 instance de la classe (`this`)
- pas d 'instruction `return` dans le constructeur

Création d 'objet

~~`new Point()`~~

`new Point(15,14)`

Constructeurs

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 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**)

Surchage des méthodes

- **surcharge** (overloading) pas limitée aux constructeurs, elle est possible pour n'importe quelle méthode
- possible de définir des méthodes possédant le même nom mais dont les arguments 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){
        ...
   }

   // 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
- l'appel **`this(...)`**
  - *fait référence au constructeur de la classe dont les arguments correspondent*
  - *ne peut être utilisé que comme première instruction dans le corps d'un constructeur, il ne peut 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 super-classe)*

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);
    }

    ...
}
```

Intérêt :

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

Possibilité de définir des constructeurs privés

Variables de classes

Point.java

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

   /**
    * ordonnée du point
    */
   private double y;

   ...

    /**
     * Compare 2 points cartésiens
     * @param p l'autre point
     * @require PointValide : p!=null
     * @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 d'imprécision

Variables de classe

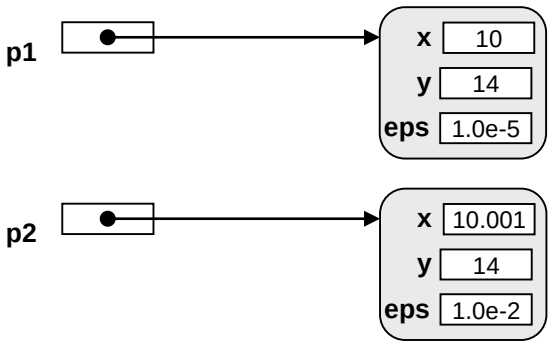
```
public class Point {
   /** abscisse du point
    */
    private double x;

   /** ordonnée du point
    */
   private double y;

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

   /** imprécision pour tests d'égalité
    */
   public void setEps(double eps) {
      this.eps = eps;
   }

   /** restitue valeur imprécision
    * @return valeur imprécision
    */
   public double getEps() {
      return eps;
   }
  ...
  /** Compare 2 points cartésiens
    * @param p l'autre point
    * @require PointValide : p!=null
    * @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 <  eps  && dy <  eps  ;
    }
    ...
}
```

1ère solution :

Ajouter une variable (un attribut) Eps à la classe avec une méthode accesseur et une méthode « modifieur »

Quels sont les problèmes liés à cette solution ?

Variables de classe



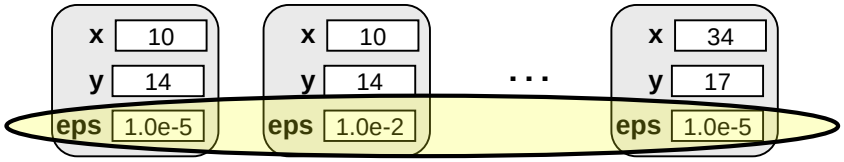

```
Point p1 = new Point(10,14);
Point p2 = new Point(10.001,14.001);
p2.setEps(10e-2);
System.out.println(p1.egale(p2));
```

→ false on utilise la précision de p1 (0.00001)

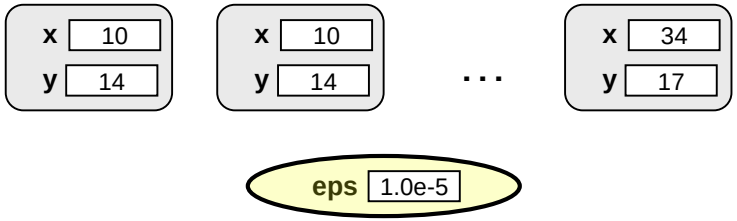
```
System.out.println(p2.egale(p1));
```

→ 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

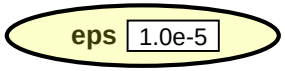


Cette information ne concerne pas une instance particulière mais l'ensemble du domaine des **`Point`**

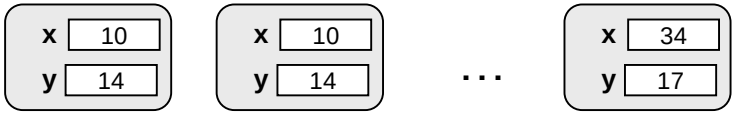


Comment représenter cet ensemble ?

Variables de classe



Valeur associée à l'ensemble des points cartésiens



Mais en Java, rien ne peut être défini en dehors d'un objet. Pas de variables globales ☺

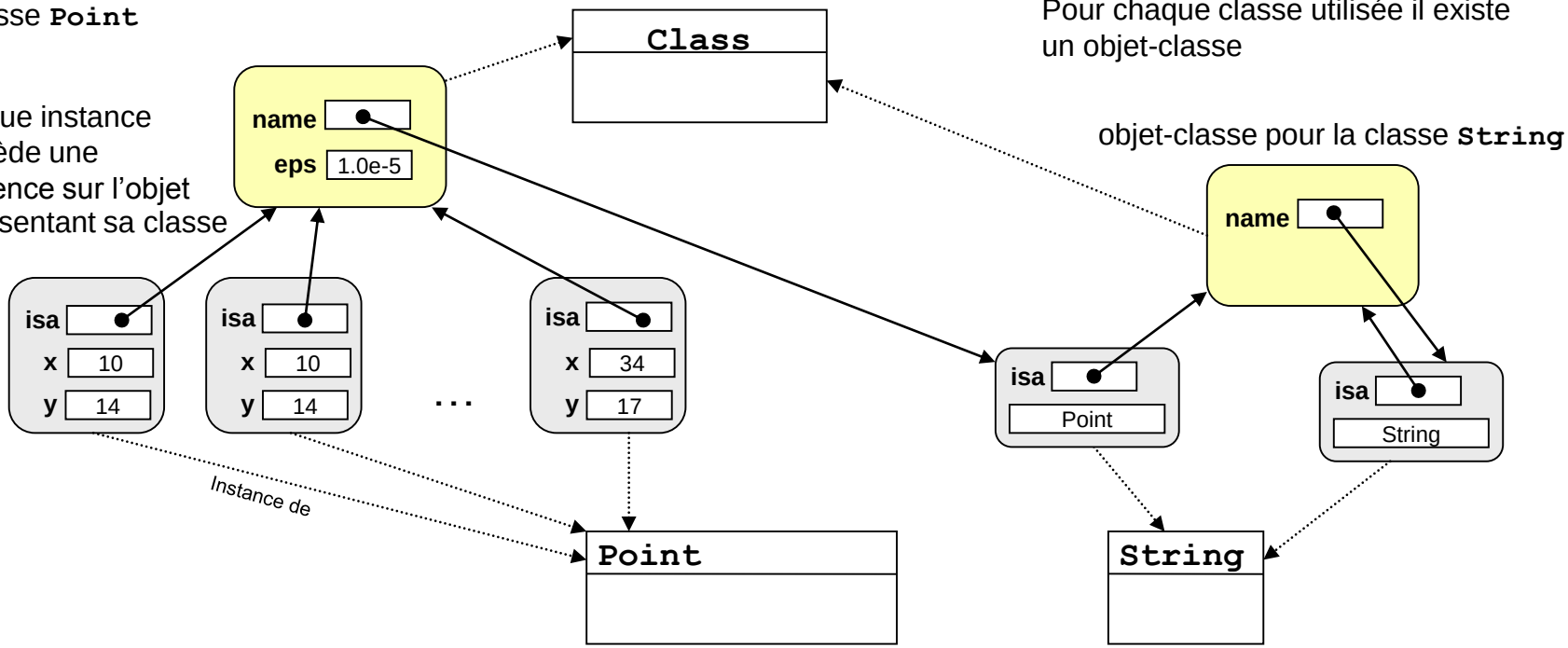
Cet objet est instance de la classe **`Class`** **`(package java.lang)`**

Un objet représente la classe **`Point`**

Pour chaque classe utilisée il existe un objet-classe

Chaque instance possède une référence sur l'objet représentant sa classe

objet-classe pour la classe **`String`**



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

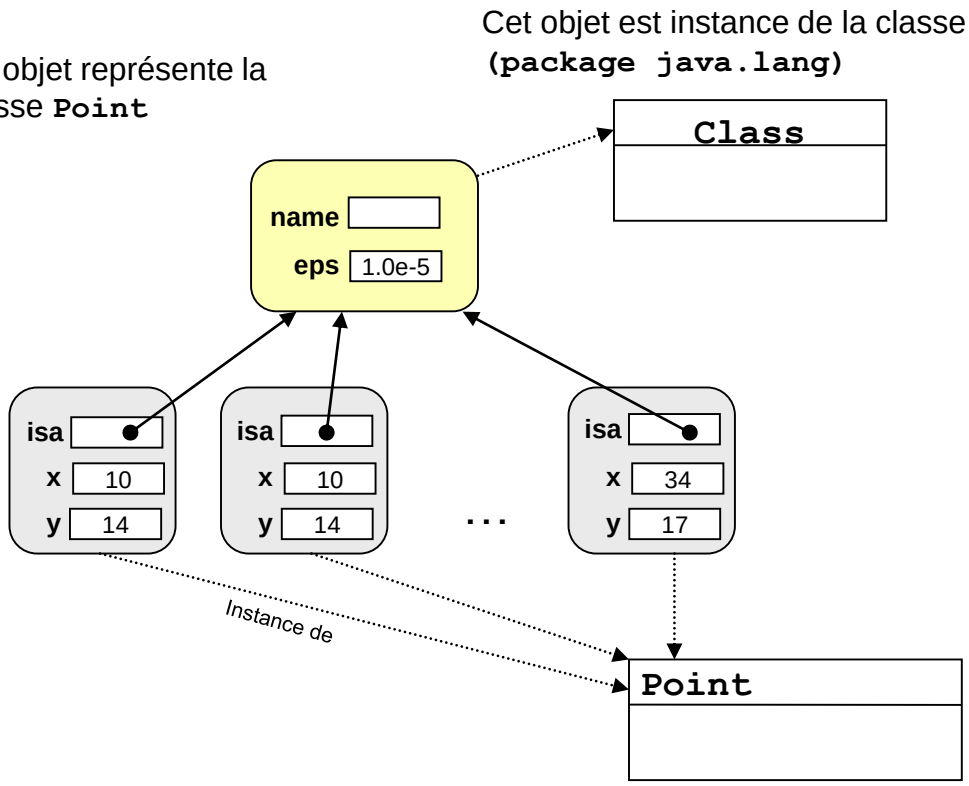
Returns a Field object that reflects the specified public member field of the class or interface represented by this Class object.

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.

…

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`**

Variable de classe

Déclaration

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

   /** ordonnée du point
    */
   private double y;

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

   /** imprécision pour tests d'égalité
    */
   public void setEps(double eps) {
      this.eps = eps;
   }

   /** restitue valeur imprécision
    * @return valeur imprécision
    */
   public double getEps() {
      return eps;
   }
  ...
  /** Compare 2 points cartésiens
    * @param p l'autre point
    * @require PointValide : p!=null
    * @return true si les points sont égaux à 1.0e-5 près
    */
    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

Variable de classe

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

   /** ordonnée du point
    */
   private double y;

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

   /** imprécision pour tests d'égalité
    */
   public void setEps(double eps) {
      this.eps = eps;
   }

   /** restitue valeur imprécision
    * @return valeur imprécision
    */
   public double getEps() {
      return eps;
   }
  ...
  /** Compare 2 points cartésiens
    * @param p l'autre point
    * @require PointValide : p!=null
    * @return true si les points sont égaux à 1.0e-5 près
    */
    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

Accès

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** ?

Variable de classe

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

   /** ordonnée du point
    */
   private double y;

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

   /** imprécision pour tests d'égalité
    */
   public void setEps(double eps) {
      this.eps = eps;
   }

   /** restitue valeur imprécision
    * @return valeur imprécision
    */
   public double getEps() {
      return eps;
   }
  ...
  /** Compare 2 points cartésiens
    * @param p l'autre point
    * @require PointValide : p!=null
    * @return true si les points sont égaux à 1.0e-5 près
    */
    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

Accès

- 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 {
   /** abscisse du point
    */
    private double x;

   /** ordonnée du point
    */
   private double y;

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

   /** imprécision pour tests d'égalité
    */
   public static tEps(double eps) {
      Point.eps = eps;
   }

   /** restitue valeur imprécision
    * @return valeur imprécision
    */
   public static getEps() {
      return eps;
   }
   ...
   /** Compare 2 points cartésiens
     * @param p l'autre point
     * @require PointValide : p!=null
     * @return true si les points sont égaux à 1.0e-5 près
     */
     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 (qui serait `this` ?)

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

```
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

- membres dont la déclaration est précédée du modifieur `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)` ...

System.out.println

???

```
System.out.println("COUCOU");
```

Classe `System` du package `java.lang`

Variable de classe (référence un objet de type `PrintStream`)

Méthode d 'instance de la classe `PrintStream` du package `java.io`

Le main()

- Le point d 'entrée pour l'exécution d 'une application Java est la méthode statique **`main`** de la classe spécifiée à la machine virtuelle
- profil de cette méthode
  `public static void main(String[] args)`
- **`String[] args`** ???
  - *args : tableau d'objets String (chaînes de caractères) contenant les arguments de la ligne de commande*

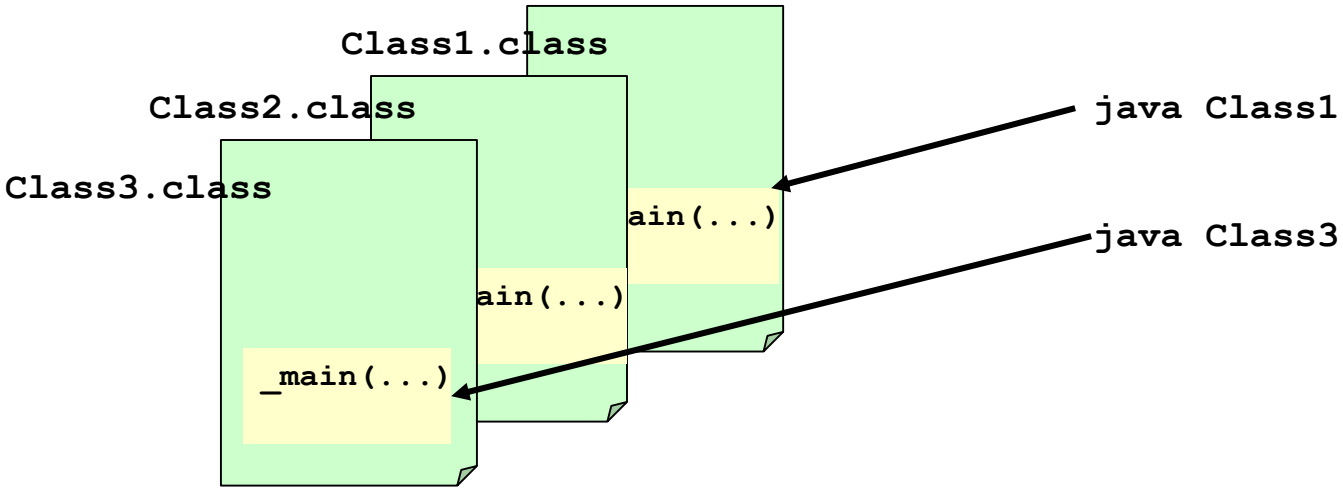
```
public class TestArgs {
 public static void main(String[] args) {
   System.out.println("nombre d 'arguments : " + args.length);
   for (int i =0; i < args.length; i++)
        System.out.println(" argument " + i + " = " + args[i]);
  }
}
```

`java TestArgs arg1 20 35.0` →

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

Le main()

- 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 une 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*

Destruction des objets

rappels

- 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 un 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 gestion explicite de la mémoire , mais programmation beaucoup plus simple et sure.

Destruction des objets

finalisation

- "ramasse miettes" gère automatiquement 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.
- méthode dite de "**finalisation**" prévue à cet effet
  - *permet de fermer les fichiers ouverts, terminer les connexions réseau... avant la destruction des objets*
- la méthode de "finalisation" :
  - *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*

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

Appel explicite au 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]
```

Destruction des objets

finalisation

- 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.*

  - *il vaut mieux éviter de définir et de compter sur des "finaliseurs" pour libérer des ressources critiques*
  - *il vaut mieux compter sur une méthode explicite de terminaison*
    - *ex : méthode close() d'un connexion JDBC (base de données)*

voir Chapter 2. Creating and Destroying Objects - Item 7
du livre *Effective Java™, Second Edition*, Joshua Bloch, Ed. Addison-Wesley Professional - 2008

http://www.codeproject.com/Articles/30593/Effective-Java