

# 2i002 - UML (light), diagramme mémoire et pointeurs

Vincent Guigue

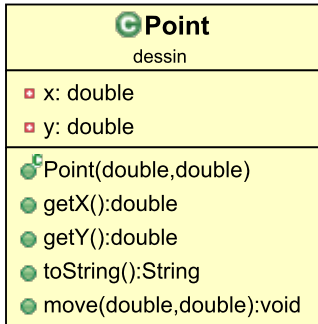


On ne programme pas pour soi-même... Mais pour les autres :

- Respecter les **codes syntaxiques** : majuscules, minuscules...
- Donner des **noms explicites** (classes, méthodes, attributs)
- Développer une **documentation** du code (cf cours javadoc)
- ... Et proposer une **vision synthétique** d'un ensemble de classes : **⇒ UML**

On ne programme pas pour soi-même... Mais pour les autres :

- Respecter les **codes syntaxiques** : majuscules, minuscules...
- Donner des **noms explicites** (classes, méthodes, attributs)
- Développer une **documentation** du code (cf cours javadoc)
- ... Et proposer une **vision synthétique** d'un ensemble de classes : **⇒ UML**

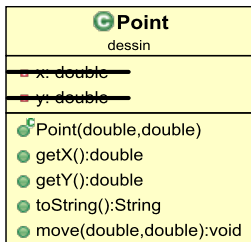


- nom de la classe
  - attributs
  - méthodes (et constructeurs)
- + code pour visualiser **public/private**
- + liens entre classes pour les dépendances (cf cours sur la composition)

# UML CLIENT vs DÉVELOPPEUR

Plusieurs types de diagrammes pour plusieurs usages :

- Pour les développeurs : représentation complète
- Pour les utilisateurs : représentation public uniquement



Idée :

Le code doit être pensé pour les autres :

- Tous les noms doivent être aussi clairs que possible
- Un diagramme plus limité est plus facile à lire

## 2 manières de voir l'UML :

- ① Outil pour une **visualisation** globale d'un code complexe
- ② Outil de **conception** / développement indépendant du langage

Dans le cadre de 2i002 : seulement l'approche ①

## Limites de l'UML :

- Vision architecte...
- Mais pas d'analyse de l'exécution du code

Que se passe-t-il lors de l'exécution du programme :

Nouveau type de représentation :

**Diagramme mémoire**

## COTÉ JVM : EXÉCUTION DU CODE

```
1 Point p = new Point(1,2);
```

Comment décrire cette ligne de code ?

```
1 Point p = new Point(1,2);
```

Comment décrire cette ligne de code ?

La **variable** `p`, de type `Point`, **référence** une **instance** dont les **attributs** `x` et `y` ont pour valeur respective 1 et 2.

Comment représenter cette ligne de code ?

```
1 Point p = new Point(1,2);
```

Comment décrire cette ligne de code ?

La **variable** `p`, de type `Point`, **référence** une **instance** dont les **attributs** `x` et `y` ont pour valeur respective 1 et 2.

Comment représenter cette ligne de code ?



- Représentation des classes sans les méthodes
- Valeur des attributs
- Type & noms des variables
- Lien de référencement



## TYPES DE BASE *vs* OBJET : SIGNIFICATION DE =

Les **types de base** et les **objets** ne se comportent pas de la même façon avec =

- Liste des **types de base** :

`int, double, boolean, char, byte, short, long, float`

```
1 double a, b;  
2 a = 1;  
3 b = a; // duplication de la valeur 1
```

⇒ Si b est modifié, pas d'incidence sur a

## TYPES DE BASE *vs* OBJET : SIGNIFICATION DE =

Les **types de base** et les **objets** ne se comportent pas de la même façon avec =

- Liste des **types de base** :

`int, double, boolean, char, byte, short, long, float`

```
1 double a, b;  
2 a = 1;  
3 b = a; // duplication de la valeur 1
```

⇒ Si b est modifié, pas d'incidence sur a

- et pour un **Objet** :

```
4 Point p = new Point(1,2);  
5 Point q = p; // duplication de la référence...  
6           // 1 seule instance !
```

## TYPES DE BASE vs OBJET : SIGNIFICATION DE =

Les **types de base** et les **objets** ne se comportent pas de la même façon avec =

- Liste des **types de base** :

**int**, **double**, **boolean**, **char**, **byte**, **short**, **long**, **float**

```
1 double a, b;  
2 a = 1;  
3 b = a; // duplication de la valeur 1
```

⇒ Si b est modifié, pas d'incidence sur a

- et pour un **Objet** :

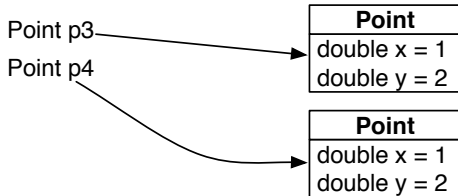
```
4 Point p = new Point(1,2);  
5 Point q = p; // duplication de la référence...  
6           // 1 seule instance !
```



2 variables, 2 références, mais 1 seule instance

# CLONAGE *vs* COPIE DE SURFACE

```
1 public static void main(String [] args) {  
2     Point p1 = new Point(1, 2);  
3     Point p2 = p1;  
4  
5     Point p3 = new Point(1, 2);  
6     Point p4 = new Point(1, 2);  
7 }
```



- Les variables p1 et p2 référencent la même instance
- p3 et p4 référencent des instances différentes

# RÉFÉRENCES & ARGUMENTS DE FONCTIONS

- Passer un argument à une fonction revient à utiliser un signe =
- ... Objets et types de base se comportent différemment !

```
1 // classe UnObjet,  
2 // (classe sans importance)  
3 ...  
4 public void maFonction1(Point p){  
5     ...  
6     p.move(1., 1.);  
7     ...  
8 }  
9  
10 public void maFonction2(double d){  
11     ...  
12     d = 3.; // syntaxe correcte  
13             // mais très moche !  
14     ...  
15 }
```

```
1 // dans le main  
2 UnObjet obj = new UnObjet();  
3  
4 Point p = new Point(1.,2.);  
5 double d = 2.;  
6  
7 obj.maFonction1(p);  
8 obj.maFonction2(d);  
9  
10 // p a pour attributs (x=2.,y=3.)  
11 // d vaut 2
```

- Quand un objet est passé en argument :  
**il n'y a pas duplication de l'instance** (simplement 2 références vers 1 instance)
- Quand un type de base est passé en argument : duplication.

## TYPES DE BASE *vs* OBJET : SIGNIFICATION DE ==

- Opérateur == : prend 2 opérandes de **même type** et retourne un boolean
- Type de base : vérification de l'égalité **des valeurs**


```
1  double d1 = 1.;  
2  double d2 = 1.;  
3  System.out.println(d1==d2); // affichage de true  
4                                //dans la console
```

## TYPES DE BASE *vs* OBJET : SIGNIFICATION DE ==

- Opérateur == : prend 2 opérandes de **même type** et retourne un boolean
- Type de base : vérification de l'égalité **des valeurs**
- Objet : vérification de l'égalité **des références**

```
1  double d1 = 1.;
2  double d2 = 1.;
3  System.out.println(d1==d2); // affichage de true
4                               //dans la console
5  Point p1 = new Point(1, 2);
6  Point p2 = p1;
7  System.out.println(p1==p2); // affichage de true
8
9  Point p3 = new Point(1, 2);
10 Point p4 = new Point(1, 2);
11 System.out.println(p3==p4); // affichage de false
```

## TYPES DE BASE *vs* OBJET : SIGNIFICATION DE ==

- Opérateur == : prend 2 opérandes de **même type** et retourne un boolean
- Type de base : vérification de l'égalité **des valeurs**
- Objet : vérification de l'égalité **des références**
-  **ATTENTION** aux classes enveloppes (qui sont des objets)

```
1  double d1 = 1.;
2  double d2 = 1.;
3  System.out.println(d1==d2); // affichage de true
4                               //dans la console
5  Point p1 = new Point(1, 2);
6  Point p2 = p1;
7  System.out.println(p1==p2); // affichage de true
8
9  Point p3 = new Point(1, 2);
10 Point p4 = new Point(1, 2);
11 System.out.println(p3==p4); // affichage de false
12 Double d3 = 1.; // classe enveloppe Double = objet
13 Double d4 = 1.;
14 System.out.println(d3==d4); // affichage de false
```



# CLASSES ENVELOPPES

Les types de base en JAVA sont doublés de *wrappers* ou classes enveloppes pour :

- utiliser les classes génériques (cf cours ArrayList)
- fournir quelques outils fort utiles

`int`, `double`, `boolean`, `char`, `byte`, `short`, `long`, `float` ⇒  
`Integer`, `Double`...

Outils

```
1 Double d1 = Double.MAX_VALUE; // valeur maximum possible
2 Double d2 = Double.POSITIVE_INFINITY; // valeur spécifique
3                                     // gérée dans les opérations
4 Double d3 = Double.valueOf("3.5"); // String ⇒ double
5 // Double.isNaN(double d), Double.isInfinite(double d)...
```

Documentation : <http://docs.oracle.com/javase/7/docs/api/java/lang/Double.html>

# CLASSES ENVELOPPES

Les types de base en JAVA sont doublés de *wrappers* ou classes enveloppes pour :

- utiliser les classes génériques (cf cours ArrayList)
- fournir quelques outils fort utiles

`int`, `double`, `boolean`, `char`, `byte`, `short`, `long`, `float` ⇒  
`Integer`, `Double`...

Outils

```
1 Double d1 = Double.MAX_VALUE; // valeur maximum possible
2 Double d2 = Double.POSITIVE_INFINITY; // valeur spécifique
3                                     // gérée dans les opérations
4 Double d3 = Double.valueOf("3.5"); // String ⇒ double
5 // Double.isNaN(double d), Double.isInfinite(double d)...
```

Documentation : <http://docs.oracle.com/javase/7/docs/api/java/lang/Double.html>

```
6 // conversions implicites = (un)boxing (depuis JAVA 5)
7 double d4 = d1;
8 Double d5 = d4;
```

# CLASSE ENVELOPPE : DES OBJETS AVANT TOUT

## Historique :

- *A l'origine* : Philosophie tout objet  
⇒ propre, mais pas pratique
- *Evolution* : plus de dérogation pour faciliter les usages  
⇒ classes enveloppes bcp moins utilisées

⇒ Classe enveloppe = objet!!!

```
1      double d1b = 1.;  
2      double d2b = d1b;  
3      double d3b = 1.;  
4      System.out.println(d1b == d2b);  
5      System.out.println(d1b == d3b);
```

# CLASSE ENVELOPPE : DES OBJETS AVANT TOUT

## Historique :

- *A l'origine* : Philosophie tout objet  
⇒ propre, mais pas pratique
- *Evolution* : plus de dérogation pour faciliter les usages  
⇒ classes enveloppes bcp moins utilisées

⇒ Classe enveloppe = objet!!!

```
1      double d1b = 1.;  
2      double d2b = d1b;  
3      double d3b = 1.;  
4      System.out.println(d1b == d2b);  
5      System.out.println(d1b == d3b);  
6      // true & true
```

# CLASSE ENVELOPPE : DES OBJETS AVANT TOUT

## Historique :

- *A l'origine* : Philosophie tout objet  
⇒ propre, mais pas pratique
- *Evolution* : plus de dérogation pour faciliter les usages  
⇒ classes enveloppes bcp moins utilisées

⇒ Classe enveloppe = objet!!!

```
1      double d1b = 1.;
2      double d2b = d1b;
3      double d3b = 1.;
4      System.out.println(d1b == d2b);
5      System.out.println(d1b == d3b);
6      // true & true
7      Double d1  = 1.;
8      Double d2  = d1;
9      Double d3  = 1.;
10     System.out.println(d1 == d2);
11     System.out.println(d1 == d3);
```

# CLASSE ENVELOPPE : DES OBJETS AVANT TOUT

## Historique :

- *A l'origine* : Philosophie tout objet  
⇒ propre, mais pas pratique
- *Evolution* : plus de dérogation pour faciliter les usages  
⇒ classes enveloppes bcp moins utilisées

⇒ Classe enveloppe = objet!!!

```
1      double d1b = 1.;
2      double d2b = d1b;
3      double d3b = 1.;
4      System.out.println(d1b == d2b);
5      System.out.println(d1b == d3b);
6      // true & true
7      Double d1  = 1.;
8      Double d2  = d1;
9      Double d3  = 1.;
10     System.out.println(d1 == d2);
11     System.out.println(d1 == d3);
12     // true & false
```

## POINTEUR null

Que se passe-t-il quand on déclare une variable (sans l'instancier) ?

```
1 Point p;
```

- p vaut null.
- On peut écrire de manière équivalente

```
1 Point p = null;
```

## POINTEUR null

Que se passe-t-il quand on déclare une variable (sans l'instancier) ?

```
1 Point p;
```

- p vaut null.
- On peut écrire de manière équivalente

```
1 Point p = null;
```

- On ne peut pas invoquer de méthode

```
1 p.move(1., 2.); // => CRASH de l'exécution :  
2                // NullPointerException
```



## POINTEUR null

Que se passe-t-il quand on déclare une variable (sans l'instancier) ?

```
1 Point p;
```

- p vaut null.
- On peut écrire de manière équivalente

```
1 Point p = null;
```

- On ne peut pas invoquer de méthode

```
1 p.move(1., 2.); // => CRASH de l'exécution :  
2                // NullPointerException
```

- N'importe quel objet peut être null et réciproquement, on peut donner null à n'importe quel endroit où un objet est attendu... Même si ça provoque parfois des crashes.

```
1 // classe UnObjet,  
2 // (classe sans importance)  
3 ...  
4 public void maFonction(Point p){  
5     ...  
6     p.move(1., 1.);  
7     ...  
8 }
```

```
1 // dans le main  
2 UnObjet obj = new UnObjet();  
3  
4 obj.maFonction(null);
```

# EXECUTION D'UNE MÉTHODE

Une méthode est exécutée sur une instance...

```
1 Point p = new Point(1,2);  
2 Point p2 = p;  
3 p.move(1.,1.);  
4 // méthode exécutée sur l'instance  
5 // (qui est référencée par p et p2)  
6 System.out.println(p2);  
7 // [x=2., y=3.]
```

⇒ Il faut toujours se représenter ce qui se passe dans la mémoire lors de l'exécution d'un programme.

## 2i002 - Égalité entre objets, Clonage d'objet

Vincent Guigue



# PROBLÉMATIQUE

- Le signe = se comporte de manière spécifique avec les objets...
- Le signe == également spécifique avec les objets...

## Vocabulaire (uniquement pour les opérations sur objets)

**new** Instanciation / Création d'instance



```
1 Point p = new Point(1,2);
```

# PROBLÉMATIQUE

- Le signe = se comporte de manière spécifique avec les objets...
- Le signe == également spécifique avec les objets...

## Vocabulaire (uniquement pour les opérations sur objets)

**new** Instanciation / Création d'instance

= Création de référence



```
1 Point p = new Point(1,2);
```

```
2 Point q = p;
```

# PROBLÉMATIQUE

- Le signe = se comporte de manière spécifique avec les objets...
- Le signe == également spécifique avec les objets...

## Vocabulaire (uniquement pour les opérations sur objets)

**new** Instanciation / Création d'instance

= Création de référence

== Egalité référentielle



```
1 Point p = new Point(1,2);
2 Point q = p;
3 Point r = new Point(1,2);
4 System.out.println((p==q) + " " + (p==r)); // true false
```

# COMMENT DUPLIQUER UNE INSTANCE ?

Idée (assez raisonnable somme toute)

Créer une nouvelle instance dont les valeurs des attributs sont identiques

- Exemple de code dans la classe Point

```
1 public class Point{
2   ...
3   public Point clone(){
4     return new Point(x, y);
5   }
6 }
```

- Usage :

```
1 // main
2 Point p = new Point(1,2);
3 Point p2 = p.clone();
```

NB : construction de nouvelle instance sans new explicite dans le main

Avant le JAVA, il y avait le C++

En C++ : la syntaxe standard = **constructeur de copie**

- Exemple de code dans la classe Point

```
1 // Constructeur de Point a partir d'un autre Point
2 public Point(Point p){
3     this.x = p.x; // this facultatif
4     this.y = p.y; // this facultatif
5 }
```

- Usage :

```
1 Point p = new Point(1,2);
2 Point p2 = new Point(p);
```

- Résultat ABSOLUMENT identique (depuis JAVA 1.5)
- Avantage du clone en JAVA : il s'agit d'une méthode standard (= + facile à lire)



# COMMENT TESTER L'ÉGALITÉ STRUCTURELLE ?

Idée (toujours assez raisonnable)

Créer une méthode qui teste l'égalité des attributs

- **Solution 1** (simple mais pas utilisée)

```
1 // Dans la classe Point
2 public boolean egalite(Point p){
3     return p.x == x && p.y == y;
4 }
5 // equivalent simplifié de
6 // if(p.x == x && p.y == y)
7 //     return true
8 // else
9 //     return false;
```

```
1 // Dans le main
2
3 Point p1 = new Point(1.,2.);
4 Point p2 = p1;
5 Point p3 = new Point(1.,2.);
6 Point p4 = new Point(1.,3.);
7
8 p1.egalite(p2); // true
9 p1.egalite(p3); // true
10 p1.egalite(p4); // false
```

- `public boolean egalite(Point p)` produit le résultat attendu
- ⚠ ATTENTION à la signature :
  - la méthode retourne un booléen
  - la méthode ne prend qu'un argument (on teste l'égalité entre l'instance qui invoque la méthode et l'argument)

- **Solution 2 : standard...** mais un peu plus complexe

## MÉTHODE STANDARD : `boolean equals(Object o)`

- `equals` existe dans tous les objets (comme `toString`)
  - mais teste l'égalité référentielle... Pas intéressant (comme `toString`)
- $\Rightarrow$  Redéfinition = faire en sorte de tester les attributs

Un processus en plusieurs étapes :

- 1 Vérifier s'il y a égalité référentielle / pointeur null
- 2 Vérifier le type de l'`Object o` (cf cours polymorphisme)
- 3 Convertir l'`Object o` dans le type de la classe (idem)
- 4 Vérifier l'égalité entre attributs

```
1      public boolean equals(Object obj) {  
2          if (this == obj) return true;  
3          if (obj == null) return false;  
4          if (getClass() != obj.getClass())  
5              return false;  
6          Point other = (Point) obj;  
7          if (x != other.x) return false;  
8          if (y != other.y) return false;  
9          return true;  
10     }
```

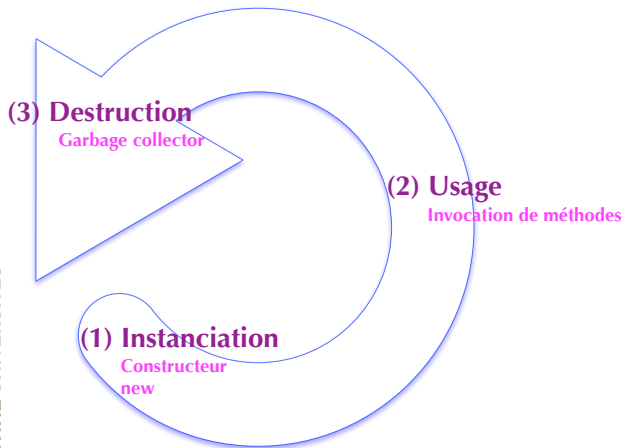
## 2i002 - Cycle de vie des objets

Vincent Guigue



# CYCLE DE VIE : DÉFINITION

Se placer du point de vue de l'objet :



(1) création d'une instance

(2) évolution de l'instance

(3) condition de destruction

# (1) INSTANCIATION

## Coté fournisseur :

*mise en route de l'objet*

Instanciation = constructeur =  
contrat d'initialisation des attributs

```
1 //Fichier Point.java
2 public class Point{
3     private double x,y;
4
5     public Point(double x2,double y2){
6         x = x2;
7         y = y2;
8     }
9 }
```

## Coté client :

*création d'une instance*

Instanciation = création d'une zone  
mémoire réservée à l'objet

```
1 //Fichier TestPoint.java
2 public class TestPoint{
3     public static void main
4         (String[] args){
5         // appel du constructeur
6         // avec des valeurs choisies
7         Point p1 = new Point(1., 2.);
8     }
9 }
```

# (1) INSTANCIATION

## Coté fournisseur :

*mise en route de l'objet*

Instanciation = constructeur =  
contrat d'initialisation des attributs

```
1 //Fichier Point.java
2 public class Point{
3     private double x,y;
4
5     public Point(double x2, double y2){
6         x = x2;
7         y = y2;
8     }
9 }
```

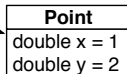
## Coté client :

*création d'une instance*

Instanciation = création d'une zone  
mémoire réservée à l'objet

```
1 //Fichier TestPoint.java
2 public class TestPoint{
3     public static void main
4         (String[] args){
5         // appel du constructeur
6         // avec des valeurs choisies
7         Point p1 = new Point(1., 2.);
8     }
9 }
```

Point p1



La variable p1, de type Point, référence un instance de Point dont les attributs ont pour valeur 1 et 2.

## (2) USAGE

- le **fournisseur** développe et garantit le bon fonctionnement des méthodes pour *utiliser* l'objet correctement,
- le **client** invoque les méthodes sur des objets pour les manipuler.

```
1 //Fichier Point.java
2 public class Point{
3     private double x,y;
4     public Point(double x2,double y2){
5         x = x2;    y = y2;
6     }
7
8     public void move(double dx,
9                     double dy){
10         x += dx; y += dy;
11     }
12     ...
13 }
```

```
1 //Fichier TestPoint.java
2 public class TestPoint{
3     public static void main
4         (String[] args){
5         // appel du constructeur
6         // avec des valeurs choisies
7         Point p1 = new Point(1., 2.);
8         p1.move(2., 3.);
9         // p1 => [x=3, y=5]
10    }
11 }
```

### (3) DESTRUCTION

- ① Un objet est détruit lorsqu'il n'est **plus référencé**
- ② La destruction est implicite (contrairement au C++) et traitée en tâche de fond (garbage collector)

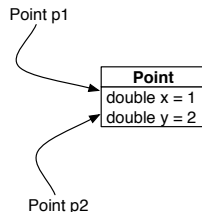


### (3) DESTRUCTION

- 1 Un objet est détruit lorsqu'il n'est **plus référencé**
- 2 La destruction est implicite (contrairement au C++) et traitée en tâche de fond (garbage collector)

- Un objet peut être **référéncé plusieurs fois...**

```
1 //Fichier TestPoint.java
2 public class TestPoint{
3     public static void main (String [] args){
4         // appel du constructeur
5         // avec des valeurs choisies
6         Point p1 = new Point(1., 2.);
7         Point p2 = p1;
8     }
9 }
```

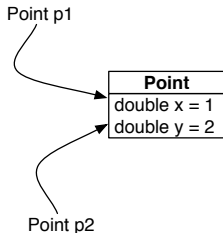


- mais quand est-il **dé-réféncé** ?

# DÉ-RÉFÉRENCEMENT D'UN OBJET

## ① Dé-référencement explicite (usage de =

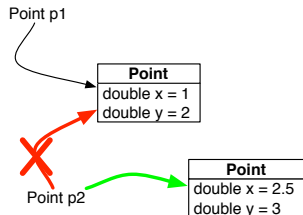
```
1 //Fichier TestPoint.java
2 public class TestPoint{
3     public static void main (String [] args){
4         // appel du constructeur
5         // avec des valeurs choisies
6         Point p1 = new Point(1., 2.);
7         Point p2 = p1;
```



# DÉ-RÉFÉRENCEMENT D'UN OBJET

## ① Dé-référencement explicite (usage de =

```
1 //Fichier TestPoint.java
2 public class TestPoint{
3     public static void main (String [] args){
4         // appel du constructeur
5         // avec des valeurs choisies
6         Point p1 = new Point(1., 2.);
7         Point p2 = p1;
8         p2 = new Point(2.5, 3.);
9     }
10 }
```



## ② Dé-référencement implicite (logique de bloc, destruction de variables $\Rightarrow$ destruction de références)

# DÉ-RÉFÉRENCEMENT D'UN OBJET

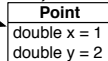
## ① Dé-référencement explicite (usage de =

```
1 //Fichier TestPoint.java
2 public class TestPoint{
3     public static void main (String [] args){
4         // appel du constructeur
5         // avec des valeurs choisies
6         Point p1 = new Point(1., 2.);
7         Point p2 = p1;
8         p2 = new Point(2.5, 3.);
9     }
10 }
```

## ② Dé-référencement implicite (logique de bloc, destruction de variables $\Rightarrow$ destruction de références)

```
1 public static void main(String [] args) {
2     {
3         Point p1 = new Point(1,2);
4         System.out.println(p1);
5     }
}
```

Point p1



# DÉ-RÉFÉRENCEMENT D'UN OBJET

## ① Dé-référencement explicite (usage de =

```
1 //Fichier TestPoint.java
2 public class TestPoint{
3     public static void main (String [] args){
4         // appel du constructeur
5         // avec des valeurs choisies
6         Point p1 = new Point(1., 2.);
7         Point p2 = p1;
8         p2 = new Point(2.5, 3.);
9     }
10 }
```

## ② Dé-référencement implicite (logique de bloc, destruction de variables $\Rightarrow$ destruction de références)

```
1 public static void main(String [] args) {
2     {
3         Point p1 = new Point(1,2);
4         System.out.println(p1);
5     }
6     System.out.println(p1);
}
```

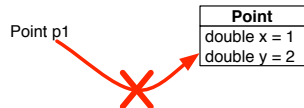
# DÉ-RÉFÉRENCEMENT D'UN OBJET

## ① Dé-référencement explicite (usage de =

```
1 //Fichier TestPoint.java
2 public class TestPoint{
3     public static void main (String [] args){
4         // appel du constructeur
5         // avec des valeurs choisies
6         Point p1 = new Point(1., 2.);
7         Point p2 = p1;
8         p2 = new Point(2.5, 3.);
9     }
10 }
```

## ② Dé-référencement implicite (logique de bloc, destruction de variables $\Rightarrow$ destruction de références)

```
1 public static void main(String [] args) {
2     {
3         Point p1 = new Point(1,2);
4         System.out.println(p1);
5     }
6     System.out.println(p1);
7     // ERREUR DE COMPILATION
8     // p1 n'existe plus ici !
```



## RETOUR SUR LA LOGIQUE DE BLOC...

- 1 le dé-référencement dépend de l'endroit où la variable est déclarée (pas de l'endroit où la variable est initialisée)
- 2 ne pas confondre la destruction d'une **variable** et la destruction d'une **instance**

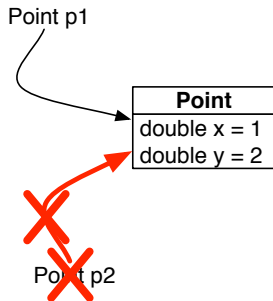
```
1 public static
2     void main(String[] args) {
3
4
5     {
6         Point p1 =
7             new Point(1,2);
8         System.out.println(p1);
9     } // destruction de
10        // la variable p1
11
12     System.out.println(p1);
13     // ERREUR DE COMPILATION
14     // p1 n'existe plus ici !
15 }
```

```
1 public static
2     void main(String[] args) {
3         Point p1; // déclaration
4                 // avant le bloc
5     {
6         // initialisation de p1
7         p1 = new Point(1,2);
8         System.out.println(p1);
9     } // pas de destruction de p1
10
11
12     System.out.println(p1);
13     // OK, pas de problème
14
15 }
```

## RETOUR SUR LA LOGIQUE DE BLOC (2)

- 1 le dé-référencement dépend de l'endroit où la variable est déclarée (pas de l'endroit où la variable est initialisée)
- 2 ne pas confondre la destruction d'une **variable** et la destruction d'une **instance**

```
1 public static
2     void main(String[] args) {
3         Point p1; // déclaration
4                 // avant le bloc
5         {
6             Point p2 = new Point(1,2);
7             // initialisation de p1
8             p1 = p2;
9             System.out.println(p1);
10        } // destruction de p2
11
12        System.out.println(p1);
13        // OK, pas de problème
14    }
```



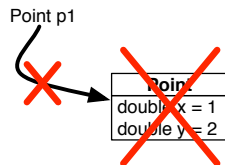
- Fin de bloc = destruction des **variables** déclarées dans le bloc
- Destruction d'instance  $\Leftrightarrow$  instance plus référencée



# DESTRUCTION DES INSTANCES

Destruction d'instance  $\Leftrightarrow$  instance plus référencée

```
1 public static void main(String[] args) {  
2     Point p1 = new Point(1,2);  
3     p1 = null;  
4 }
```

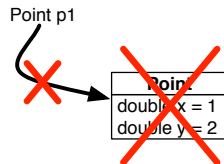


- Pas besoin d'expliquer comment détruire un objet ( $\neq$  C++)
- Le **Garbage Collector** planifie la destruction

# DESTRUCTION DES INSTANCES

Destruction d'instance  $\Leftrightarrow$  instance plus référencée

```
1 public static void main(String[] args) {  
2     Point p1 = new Point(1,2);  
3     p1 = null;  
4 }
```



- Pas besoin d'expliquer comment détruire un objet ( $\neq$  C++)
- Le **Garbage Collector** planifie la destruction

```
1 public static void main(String[] args) {  
2     ...  
3     for(int i=0; i<10; i++){  
4         // optimisation possible:  
5         // réutilisation de la mémoire allouée  
6         Point p1 = new Point((int)(Math.random()*10),  
7                               (int)(Math.random()*10));  
8         ...  
9     }  
10 }
```

- Appel explicite au garbage collector (pour libérer la mémoire) :

```
1 System.gc();
```

## LE MOT DE LA FIN...

... sur un exemple parlant :

```
1 Point p = new Point(1,2);  
2 Point p2 = new Point(3,4);  
3 // Point p3 = p; // différence avec et sans cette ligne  
4 p = p2;
```

## LE MOT DE LA FIN...

... sur un exemple parlant :

```
1 Point p = new Point(1,2);  
2 Point p2 = new Point(3,4);  
3 // Point p3 = p; // différence avec et sans cette ligne  
4 p = p2;
```

- Cas 1 : ligne commentée.
  - L'instance Point(1,2) **est détruite** à l'issue du re-référencement de p...
  - ... de toutes façons, cette instance était inaccessible.

## LE MOT DE LA FIN...

... sur un exemple parlant :

```
1 Point p = new Point(1,2);
2 Point p2 = new Point(3,4);
3 // Point p3 = p; // différence avec et sans cette ligne
4 p = p2;
```

- Cas 1 : ligne commentée.
  - L'instance Point(1,2) **est détruite** à l'issue du re-référencement de p...
  - ... de toutes façons, cette instance était inaccessible.

```
1 Point p = new Point(1,2);
2 Point p2 = new Point(3,4);
3 Point p3 = p; // différence avec et sans cette ligne
4 p = p2;
```

- Cas 2 : ligne dé-commentée
  - L'instance Point(1,2) est **conservée**...
  - On y accède par la variable p3