POO-IG

Programmation Orientée Objet et Interfaces Graphiques

Cristina Sirangelo
IRIF, Université de Paris
cristina@irif.fr

Exemples et matériel empruntés :

* Core Java - C.Horstmann - Prentice Hall Ed.
* POO in Java - L.Nigro & C.Nigro - Pitagora Ed.

Rappels :
Éléments de base de Java

Pre-requis

- On suppose la maîtrise des éléments de bases du langage Java :
  - Variables
  - Structures de contrôle
  - Commentaires
  - Types primitifs et opérateurs
  - Chaînes de caractères
  - Tableaux
  - Fonctions / Procédures
  - Entrées-sorties de base

- On suppose la maîtrise, mais on revoit :
  - Classes, objets, champs et méthodes
  - Règles de visibilité des noms

Pour commencer…

- Classes :
  En Java tout le code est dans des classes
  - une **classe** contient des **méthodes** (=fonctions) et des **variables**

- Exemple de méthode :

```
public static void affiche (String s){
  System.out.print(s + " ");
}
```

- Méthode main:

```
public static void main (String[] args)
```

Méthode main

```
//Test.java
class Test {
    public static void affiche (String s){
      System.out.print(s + " ");
    }
    public static void main(String[] args) {
        for(String a : args){
            affiche (a);
        }
    }
}
```

Main : Point d'entrée du programme: unique code exécuté par la JVM

Remarque : `for(String a : args)` est la boucle foreach en Java

Nombre variable d'arguments…

- Depuis Java 5 une méthode peut avoir un nombre variable d'arguments : `Type ...`

```
public static void affiche (String ... list){
    //list peut être manipulé comme un tableau
    for(String item : list)
            System.out.print(item + " ");
}
```

- Utilisation :

```
affiche("un", “deux","trois");
```

- ou

```
String [] l = {"un", “deux”,”trois”};
affiche (l);
```

Déclaration de variables

- Toute variable doit être initialisée avant d'être utilisée :

```
int j;
j = 5;
int i = 5;
```

possible (depuis java 10) si le type
peut être inféré de la valeur affectée :

```
var i = 5;
var s = "Bonjour";
```

Classes et Objets

Classes et objets

- Classe : définit
  - des données (variables dites champs)
  - des fonctions (méthodes) agissant sur ces données
- Classe <-> type de données
  - décrit les caractéristiques communes à une famille d'objets

- Objet : élément (instance) d'une classe avec un état
  - état de l'objet : valeur particulière des variables de la classe
  - chaque objet d'une classe peut être manipulé par les méthodes de sa classes
  - l'invocation d'une méthode peut changer l'état de l'objet

- Une méthode ou une variable peut être
  - de classe = partagée par toutes les instances de la classe ou
  - d'instance = dépendant de l'instance

Définition d'une classe

```
class ClassName {
  champs
  blocs d'initialisation
  constructeurs
  methodes
  classes / interfaces internes
}
```

champs, méthodes et classes /interfaces internes sont appelés membres

- Exemple

```
class Point {
  //champs
  private double x, y;
  //constructeurs
  public Point(double pX, double pY)
  { x = pX; y = pY;}
  // methodes
  public void deplace (double newX, double newY)
  { x = newX; y = newY; }
  public double getX () {return x;}
  public double getY () {return y;} …
}
```

Création d'objets

- Un objet d'une classe est créé avec l'opérateur `new`
- Un constructeur est invoqué automatiquement par l'opérateur new et reçoit les paramètres passés

```
class Point { … }
public class Geometry {
 public static void main( String[] args ){
    Point p = new Point(5,7);
    // p fait référence à un objet de classe Point
    //p.x vaut 5 , p.y vaut 7
    Point q = new Point(5,7);
    // q fait référence à un autre objet de classe Point
    // q.x vaut 5 , q.y vaut 7
    …
 }
}
```

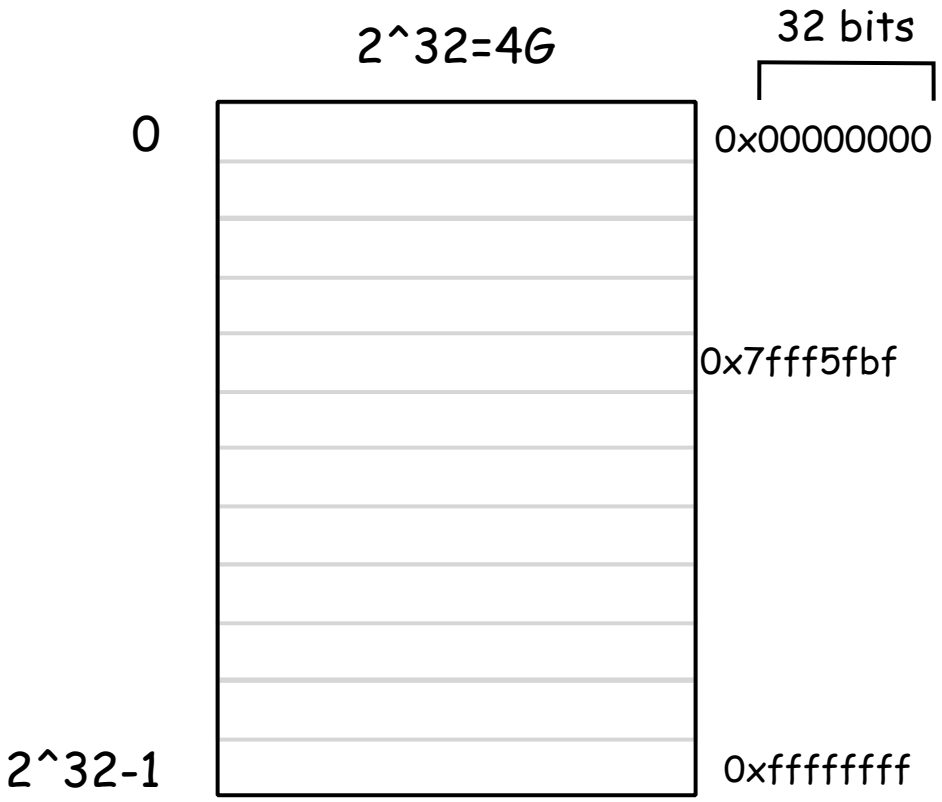
Objets et mémoire

- L'exécution d'un programme OO (par la machine virtuelle) peut être décrite en terme d'accès, vie et mort des objets en mémoire

- Pour comprendre l'exécution, il faut maîtriser les notions suivantes
  - PC, pile d'exécution et « frame »
  - gestion de la mémoire : mémoire statique et tas
  - notions de type et de référence

Espace mémoire

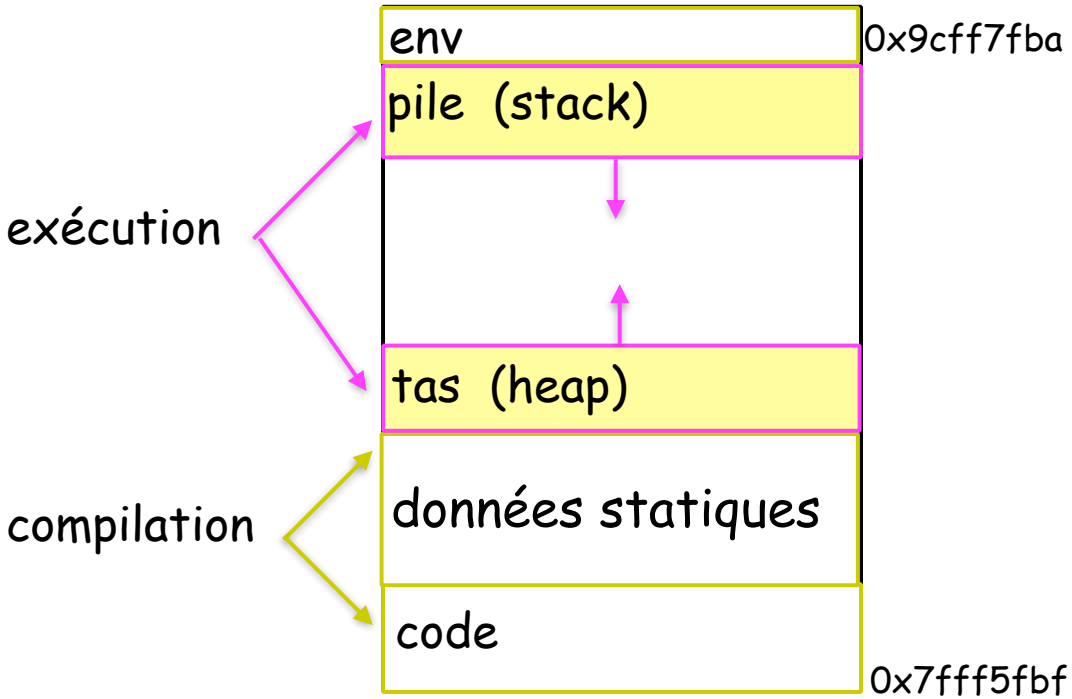
- Espace mémoire pour un processeur : une suite d'allocations de mémoire identifiées par des adresses



- adresse : nombre entier qui identifie une "position" dans la mémoire
(e.g. le nombre de bytes depuis le début
- byte addressing)

La mémoire d'un programme

- Mémoire allouée pour un programme (modèle similaire pour la machine virtuelle)

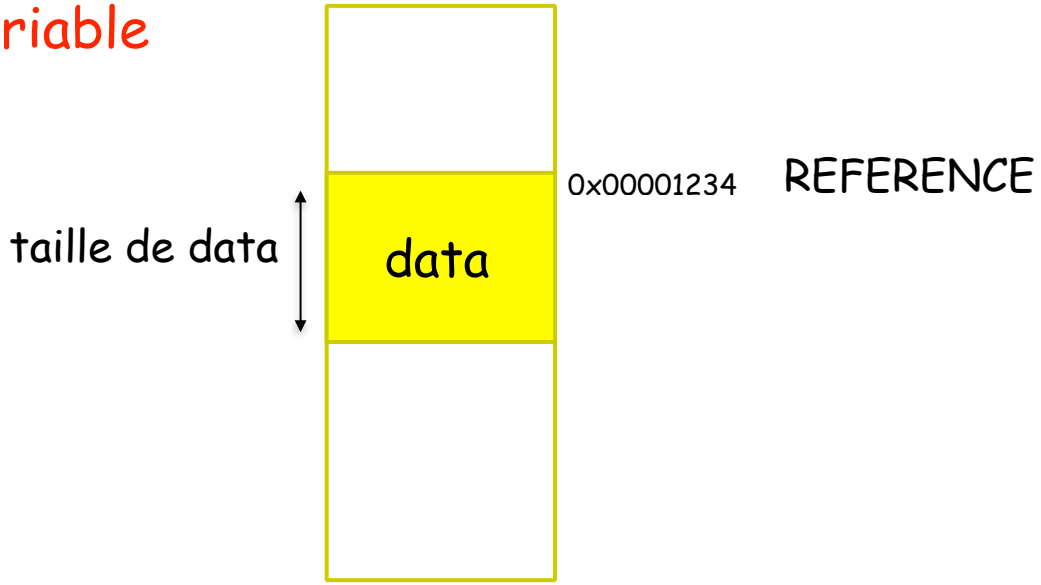


- déterminé à la compilation : (taille « fixe »)
  - code programme
  - données statiques
- déterminé à l'exécution : (taille dynamique)
  - pile d'exécution
  - zone d'allocation des variables dans le tas

La mémoire d'un programme

- Quand une variable est allouée, un emplacement mémoire d'une taille appropriée est réservé dans la mémoire du programme
- Pour récupérer les données de la variable il est suffisant de connaître son adresse de début (référence) et sa taille
- La taille est en général déterminée par le type de la variable

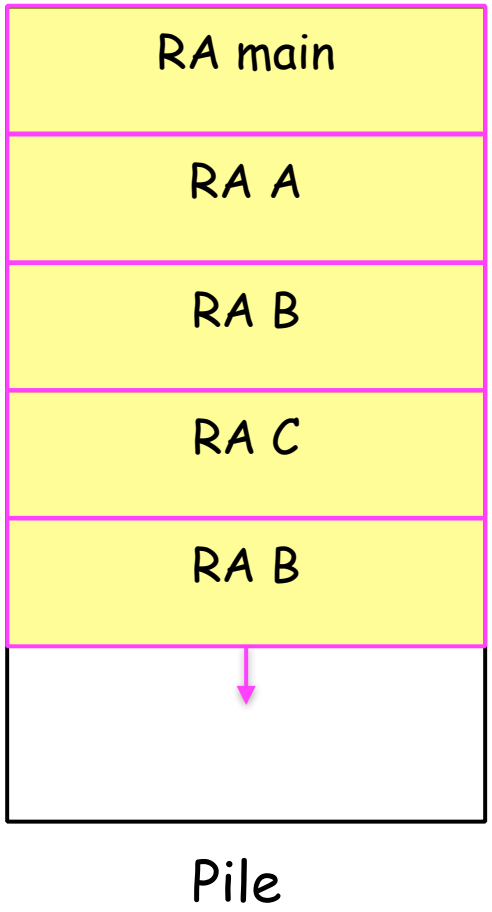
Allocation d'une variable



- Suivant les cas, l'allocation peut avoir lieu sur la pile ou dans le tas

La pile d'exécution

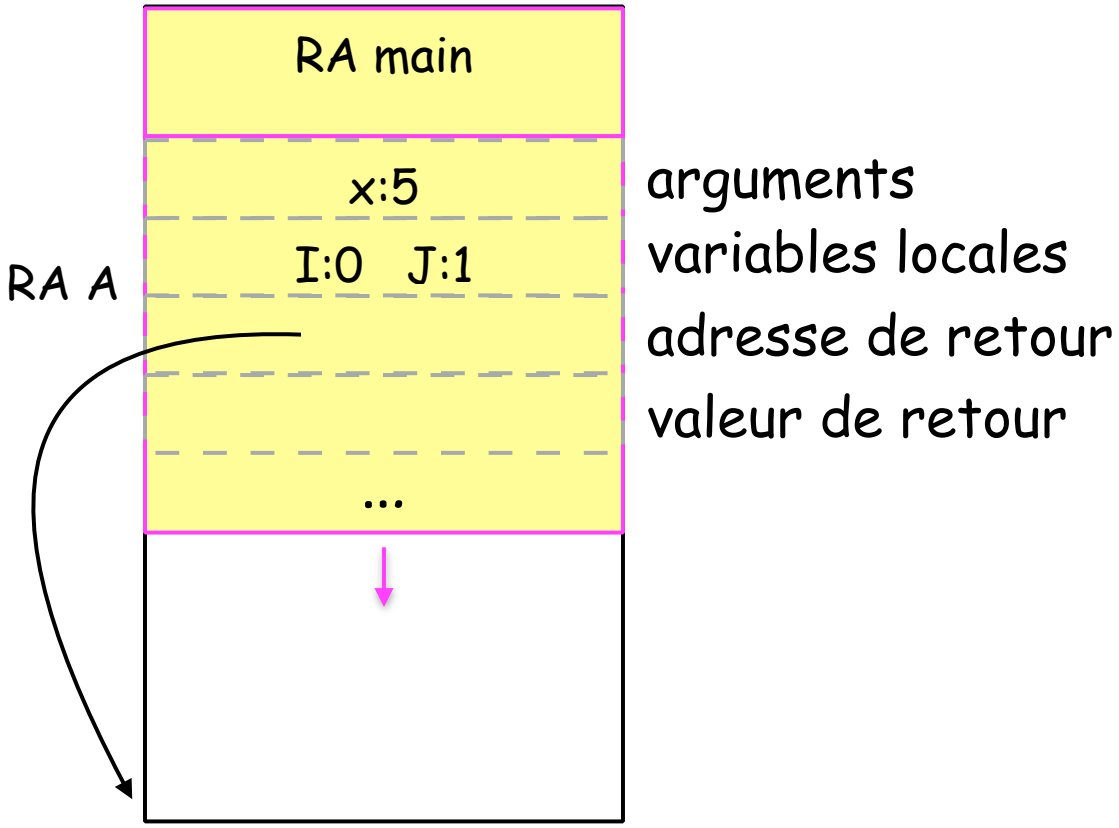
- Quand une fonction (méthode) est appelée, un nouvel espace (dit record d'activation ou <<frame>>) est empilé sur la pile d'exécution
- Ensuite compteur de programme (PC) -> l'adresse de la fonction




```
main {
… A(5); B(2);…
}
int A(int x) {
    int I=0; int J=1;  … B(3); …
}
int B(int y) {
    int J=2; … C(); … A(2); …
}
int C() {
    int K=0; … B(4); …
}
```

main calls A calls B calls C calls B

Frame (RA)




```
main {
     … A(5); B(2);
}
int A(int x) {
     int I=0; int J=1;
     … B(3); …
}
```

main calls A

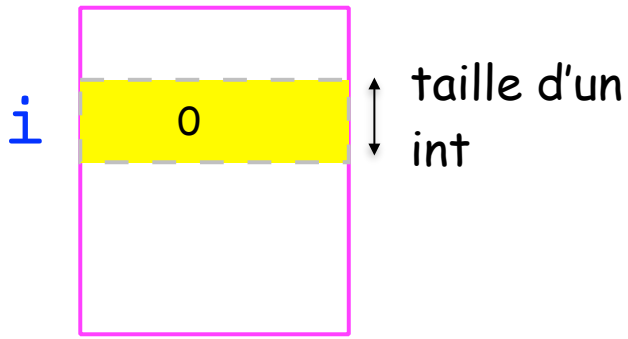
- Adresse de retour : adresse de l'instruction à exécuter quand la fonction termine
- Quand la fonction A() termine, RA A est dépilé

Objets / variables de types primitifs

- À différence des variables de type primitif (int, float, double, …) la valeur d'une variable objet n'est pas l'objet lui même mais une référence à l'objet

Type primitif

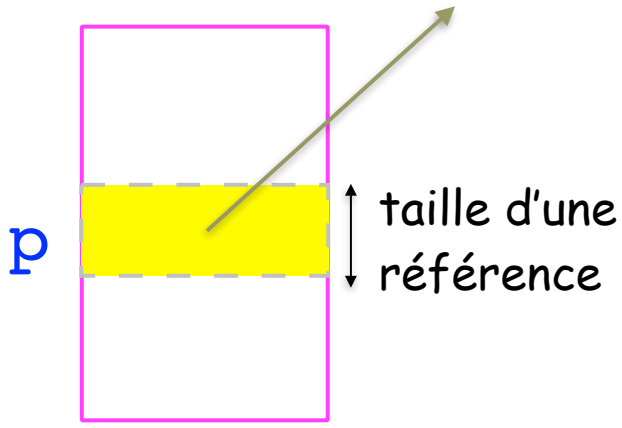
```
int i=0;
```



RA dans la pile

Objet

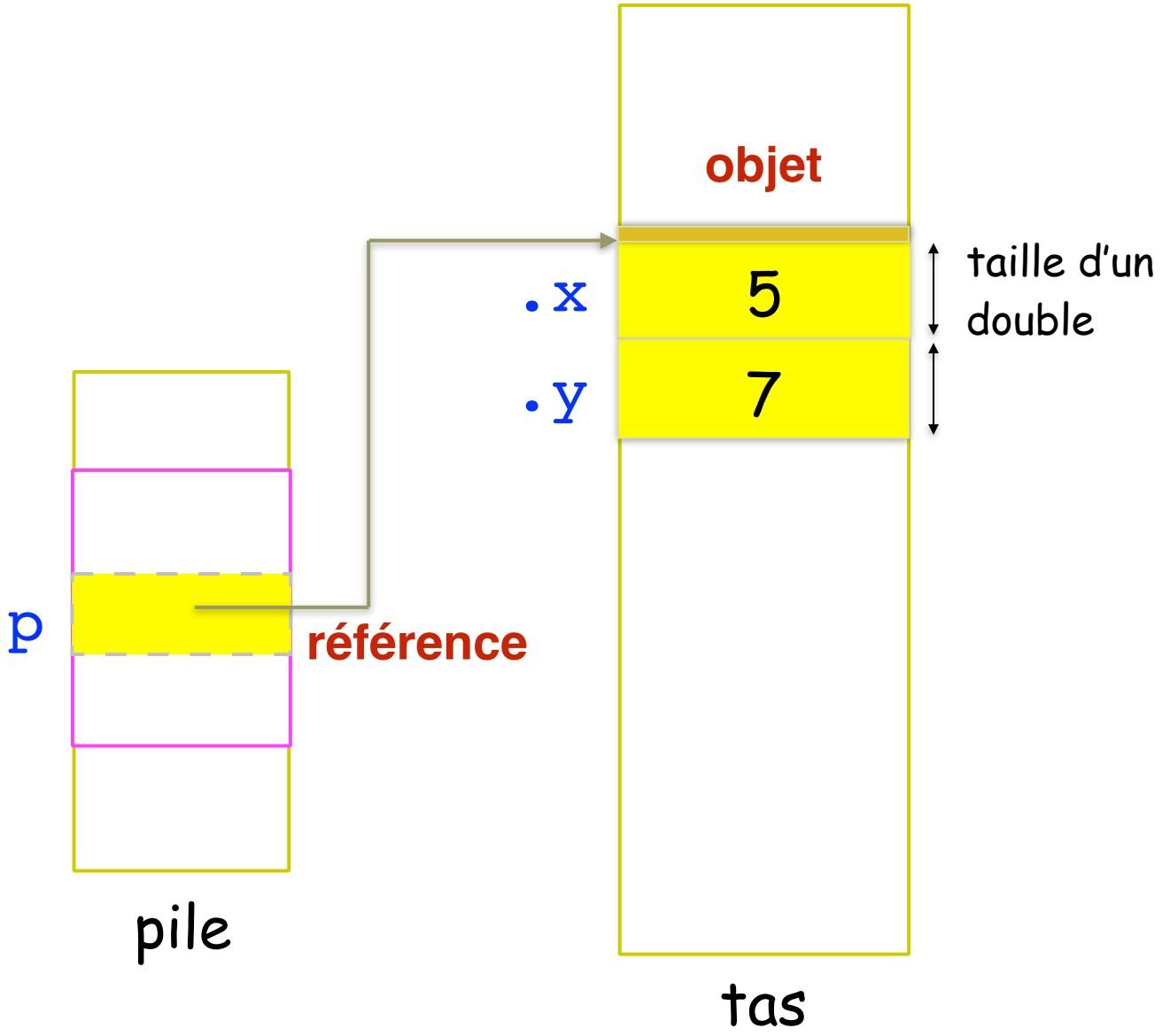
```
class Point{
  private
  double x, y;
   …
};
…
Point p =
new Point (5,7);
```



RA dans la pile

Références et objets

```
class Point{
  private
    double x, y;
  …
};
…
Point p =
new Point (5,7);
```



- L'objet lui même est créé dans le tas par l'opérateur new
- Le constructeur initialise ses champs

Objets et invocation de méthodes

- Chaque objet d'une classe peut être manipulé avec les méthodes visibles de la classe (notation '.')

```
class Point { … }
public class Geometry {
 public static void main( String[] args ){
    Point p = new Point(5,7);
    Point q = new Point(5,7);
    p.deplace(3.567,-4.6789);

    …
 }
}
```

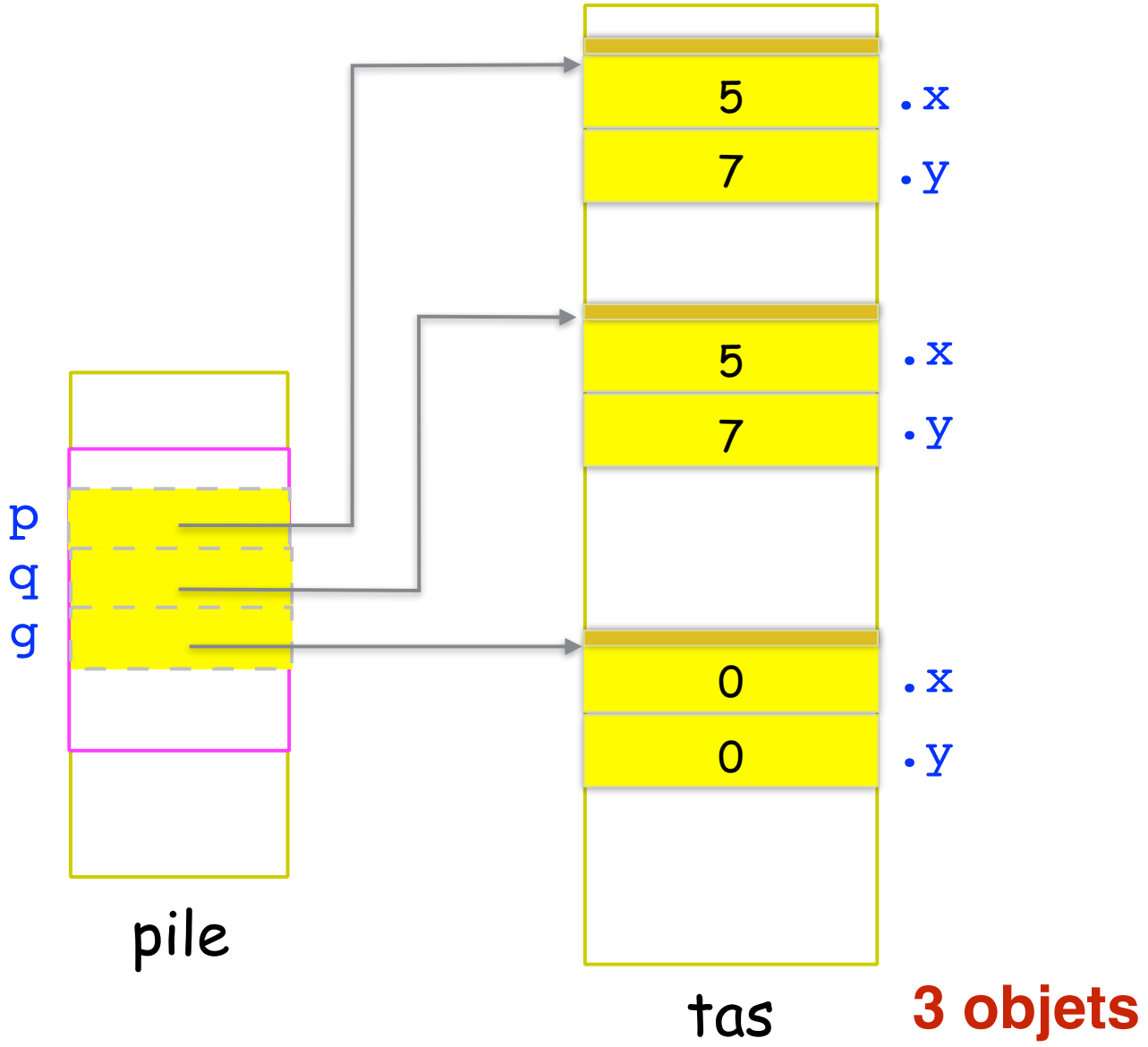
Variables d'instance

- Chaque objet de classe `Point` a sa propre copie des champs `x` et `y` (variables d'instance)
- => Il peut être manipulé indépendamment des autres
- tous les champs d'une classe sont par défaut d'instance

Variables d'instance

```
class Point{
  private
   double x, y;…
};
…
Point p =
new Point (5,7);
Point q =
new Point (5,7);
Point g =
new Point (0,0);
```

```
System.out.println
(p.getX()); // 5
System.out.println
(g.getX()); //0
```



Méthodes d'instance

- Chaque méthode d'instance opère sur l'objet sur lequel elle est invoquée :

```
p.deplace(3.567,-4.6789);
```

  modifie l'objet référencé par p

- Obtenu par le passage du paramètre implicite `this`

```
Class Point {
  …
  public void deplace (double newX, double newY)
  { this.x = newX; y = newY; }
  …
}
```

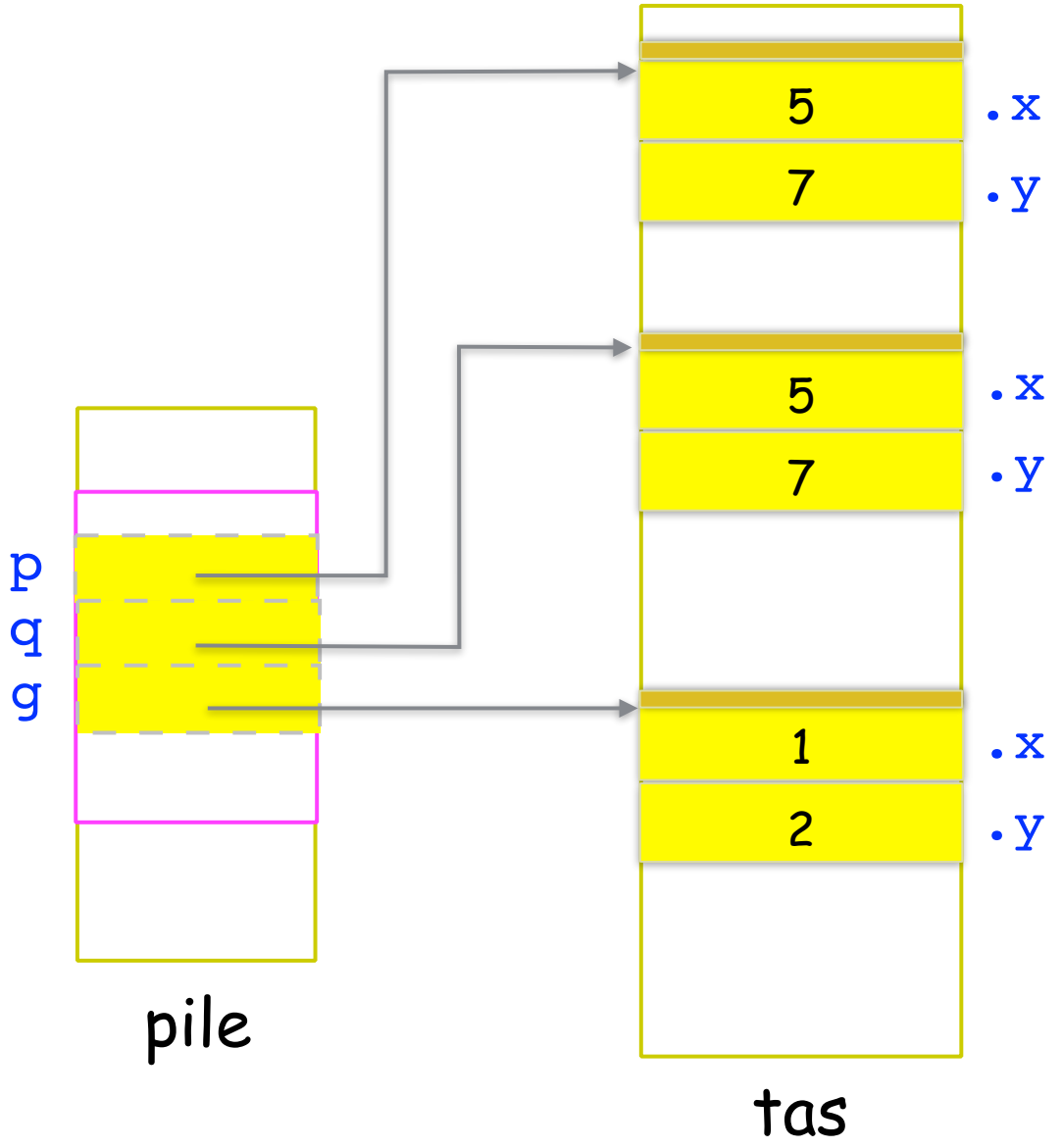
  - `this` : une référence à l'objet sur lequel la méthode est invoquée
  - `this.x` équivalent à `x` dans le corps des méthodes

- Les méthodes d'une classe sont par défaut d'instance

Méthodes d'instance

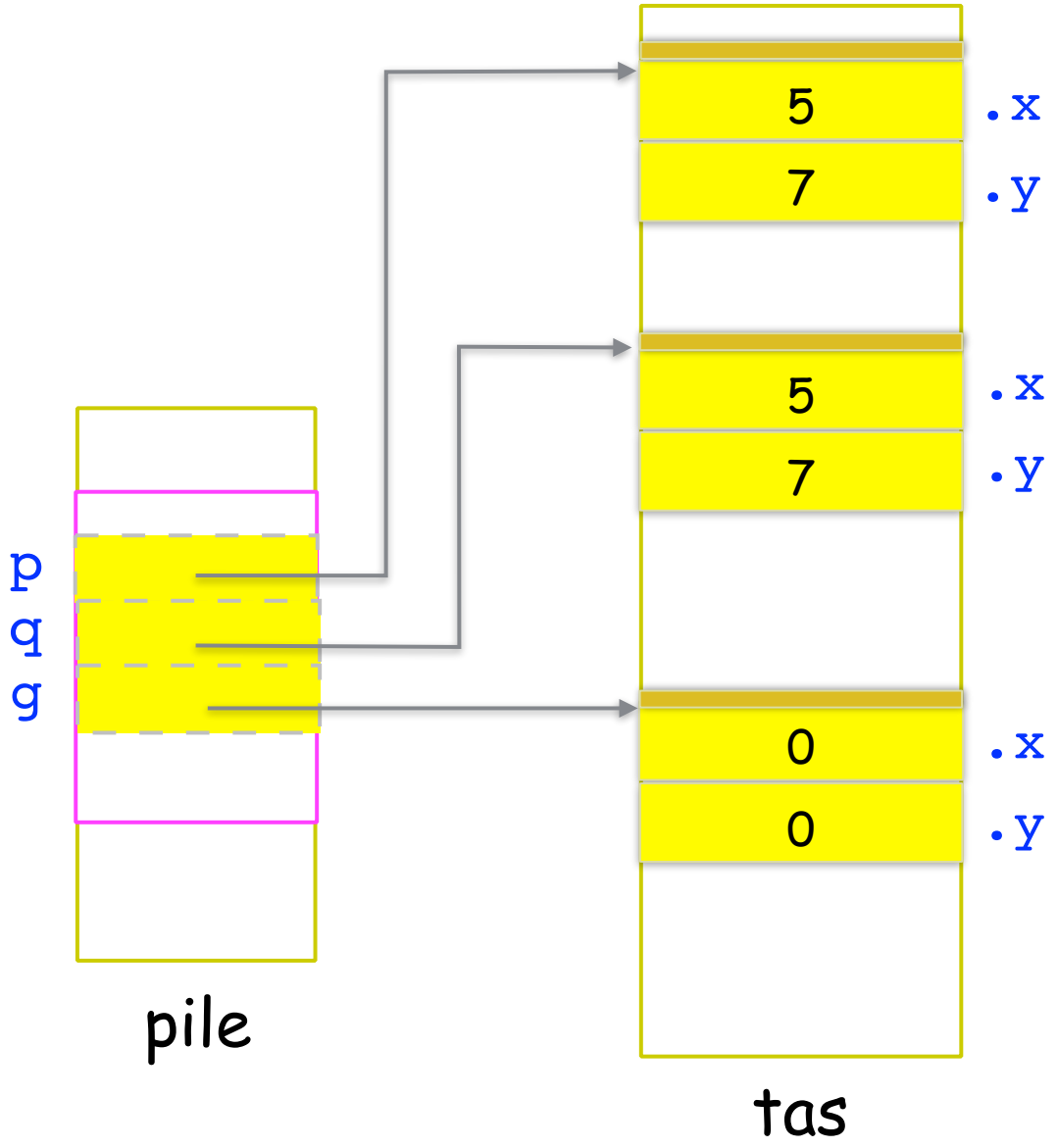
```
class Point{
  private
   double x, y;…
};
…
Point p =
new Point (5,7);
Point q =
new Point (5,7);
Point g =
new Point (0,0);
```

g.deplace (1,2)



Références et objets : test d'égalité

```
class Point{
  private
   double x, y;…
};
…
Point p =
new Point (5,7);
Point q =
new Point (5,7);
Point g =
new Point (0,0);
```



- == teste l'égalité des références

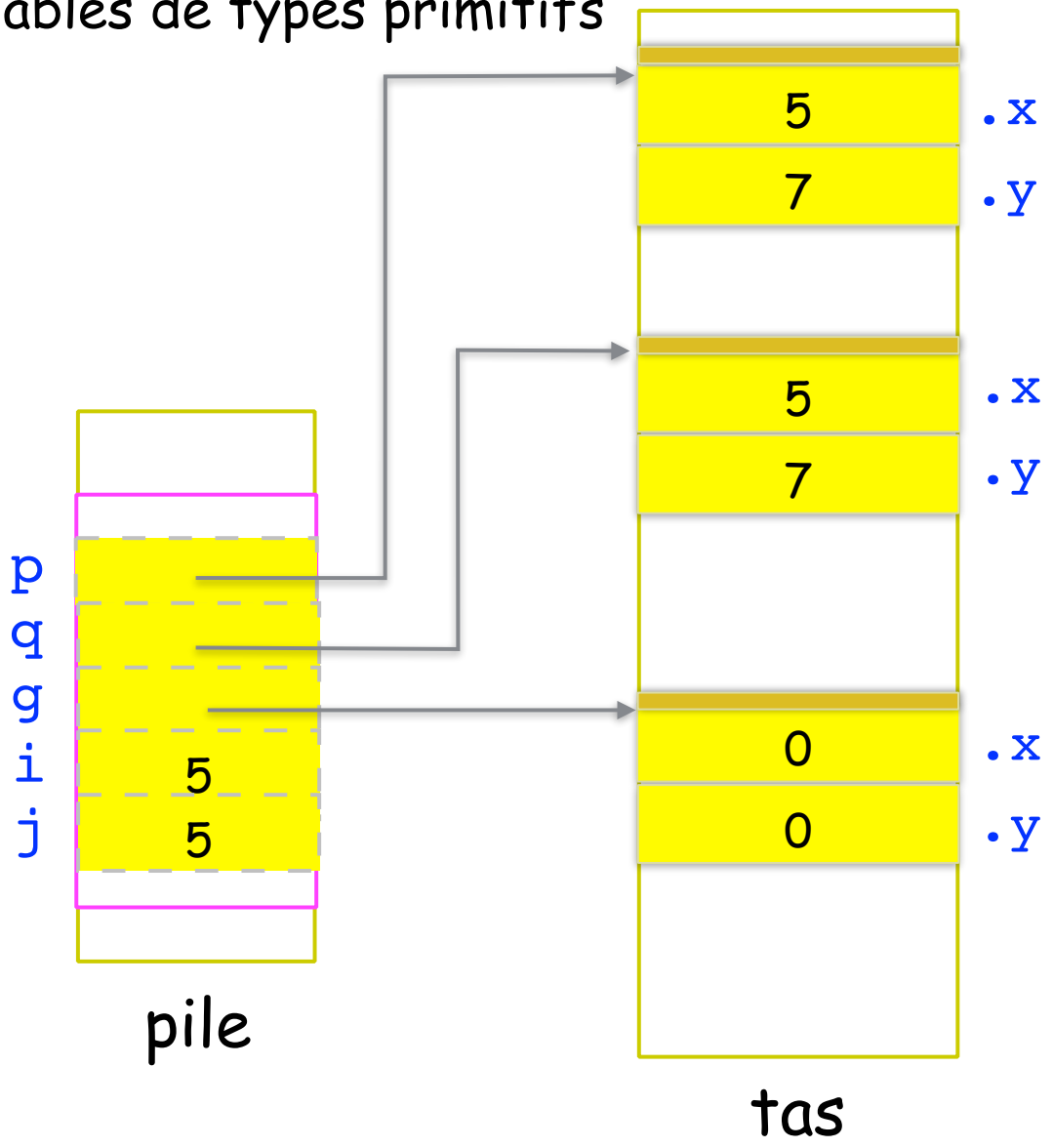
p == q //false

Références et objets : test d'égalité

- Différence avec les variables de types primitifs

```
class Point{
  private
    double x, y;
    …
};
…
Point p =
new Point (5,7);
Point q =
new Point (5,7);
Point g =
new Point (0,0);
int i = 5;
int j = 5;
```

p == q //false
i == j //vrai

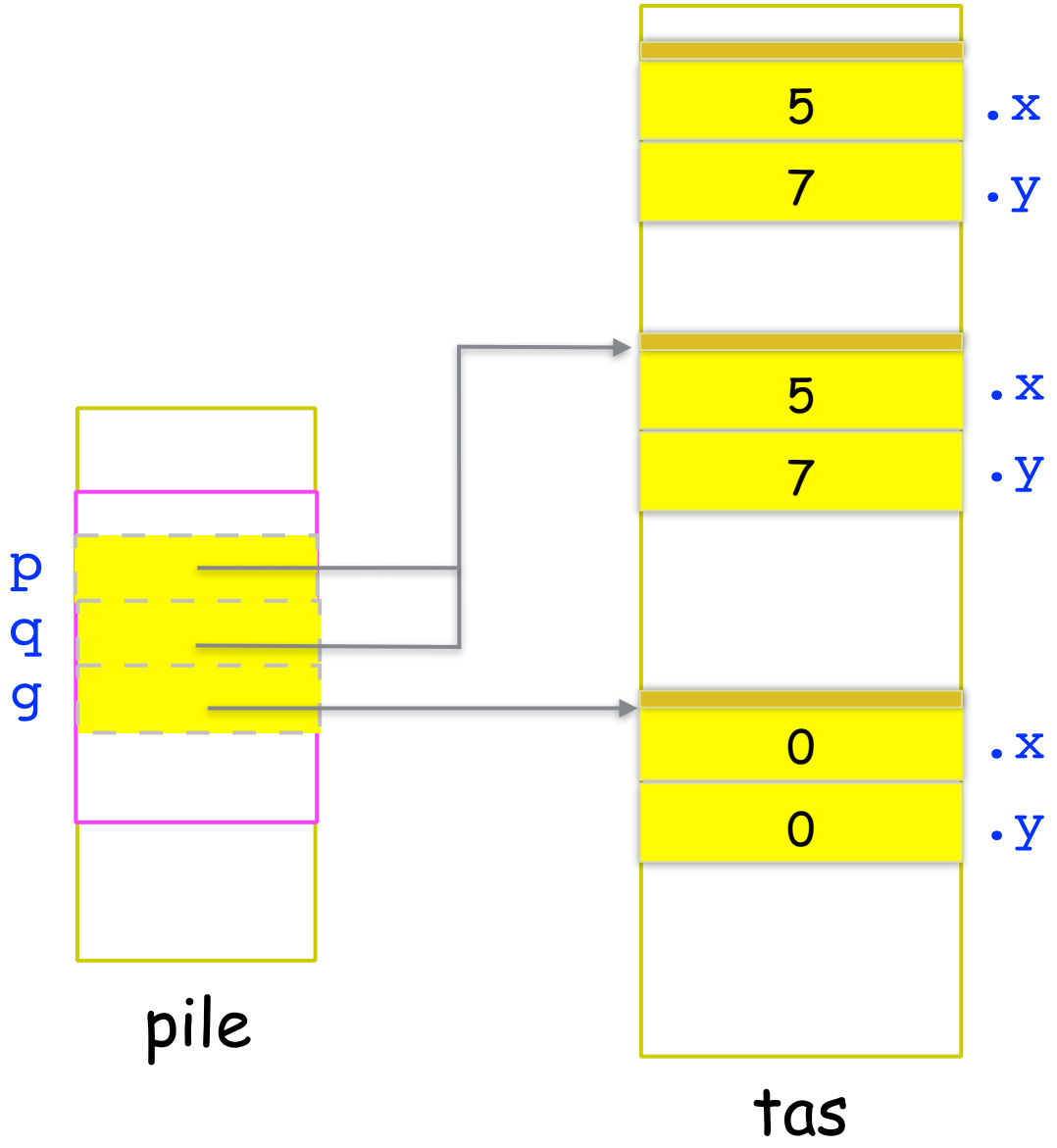


Références et objets : test et affectations

```
class Point{
  private
   double x, y;…
};
…
Point p =
new Point (5,7);
Point q =
new Point (5,7);
Point g =
new Point (0,0);
```

p = q;
p== q //true

- = affecte les références

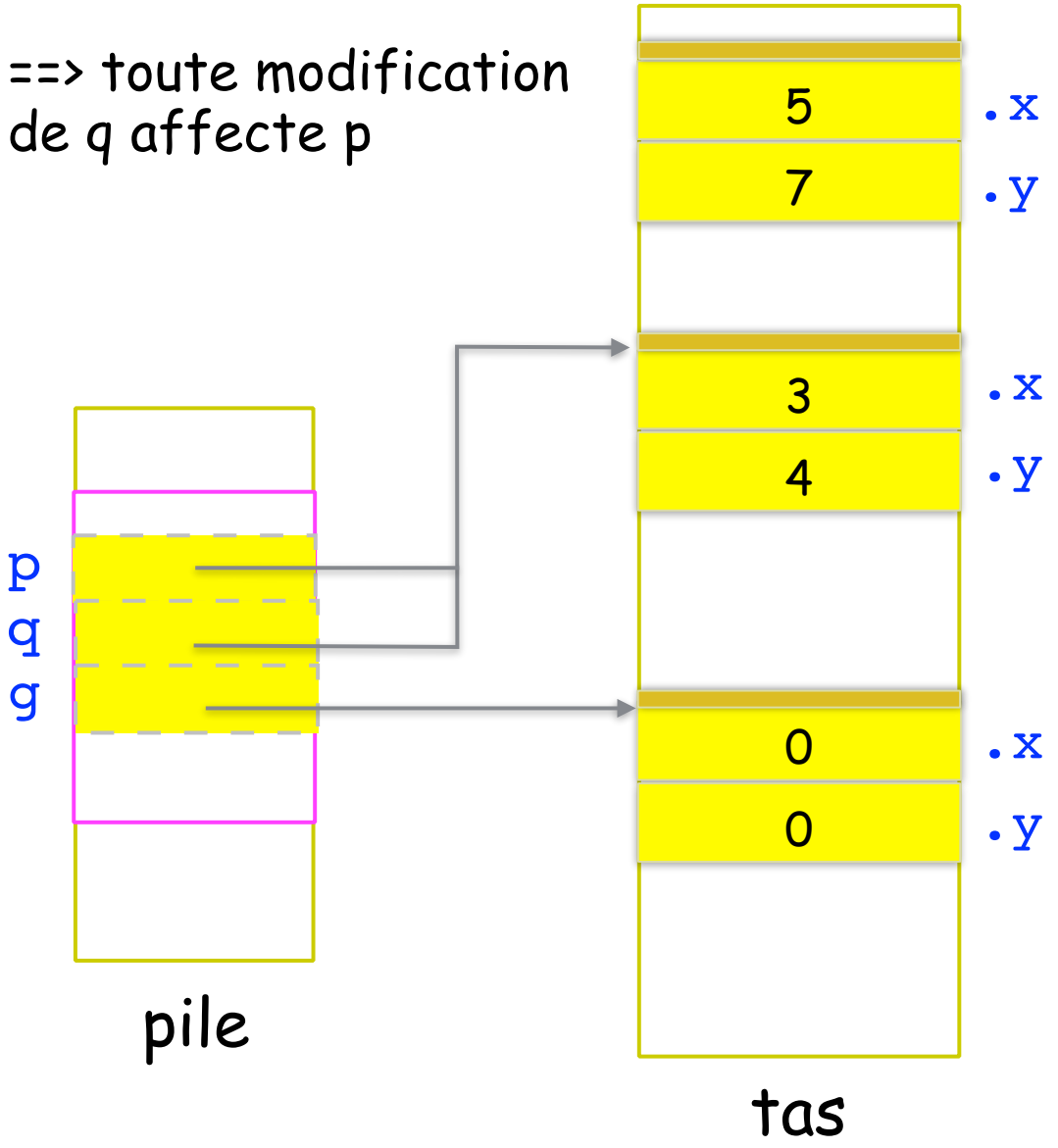


Références et objets : test et affectations

```
class Point{
  private
   double x, y;…
};
…
Point p =
new Point (5,7);
Point q =
new Point (5,7);
Point g =
new Point (0,0);
```

```
p = q;
q.deplace (3,4);
System.out.println
(p.getX()); // 3
```

- ==> toute modification de q affecte p



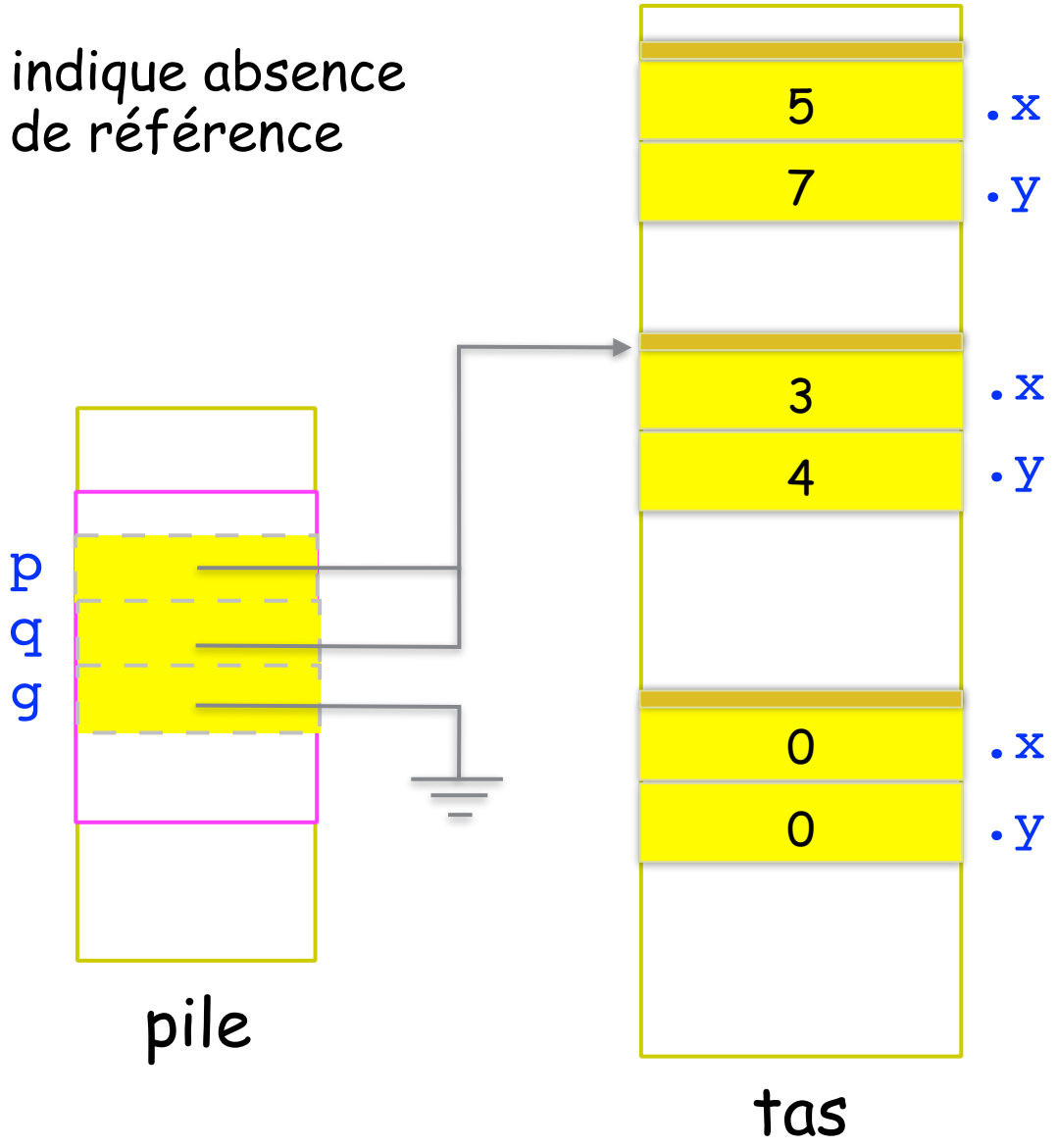
Références et objets : la constante null

```
class Point{
  private
   double x, y;
   …
};
…
Point p =
new Point (5,7);
Point q =
new Point (5,7);
Point g =
new Point (0,0);
```

g = null;

g.get(),
g.deplace(…), …
génèrent une erreur

- indique absence de référence

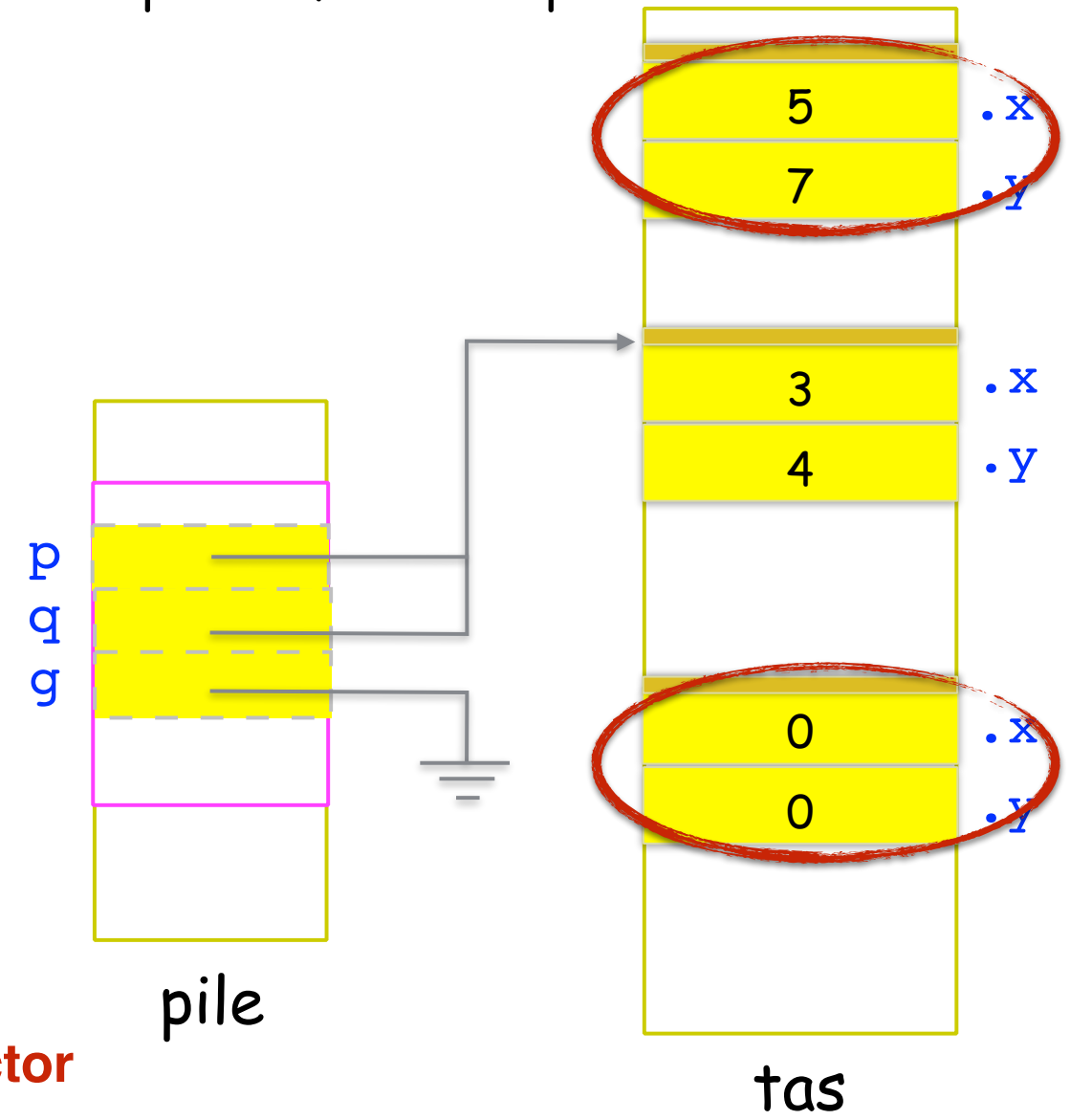


Références et objets : garbage collector

garbage : objets qui ne sont plus référencés par aucune variable

```
class Point{
  private
   double x, y;…
};
…
Point p =
new Point (5,7);
Point q =
new Point (5,7);
Point g =
new Point (0,0);
```

p = q; g = null;



Le "garbage" est détruit plus tard par un processus de Java : le **garbage collector**

Champs "static" (variables **de classe**)

- Tout membre d'une classe peut être déclaré `static`
- Une membre `static` est partagé par tous les objets de la classe
- Exemple : un compteur d'objets

```
class Point {
  //champs
  private double x, y;
  private static int compteur = 0;
  //constructeurs
  public Point(double pX, double pY)
  { x = pX; y = pY; compteur++;}
  // methodes
  …
}
```

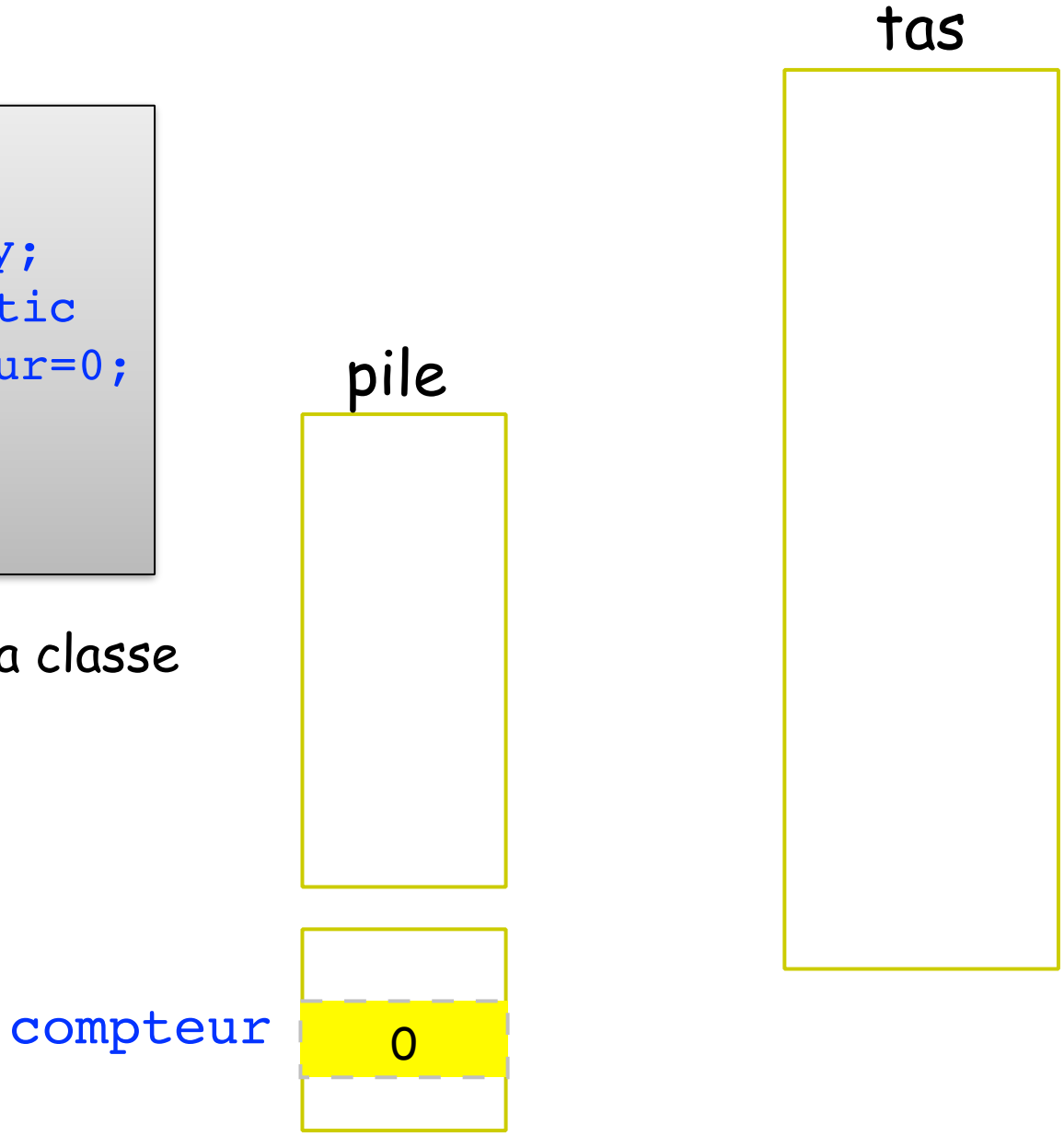
- `x, y` :
  variables d'instance
  - chaque objet de la classe a sa propre copie de `x` et `y`
- `compteur` :
  variable de classe
  - il y a une seule variable `compteur` , partagée par tous les objets de la classe

- Quand la classe est chargée en mémoire, les membres static sont alloués et initialisés

Champs "static" (variables de classe)

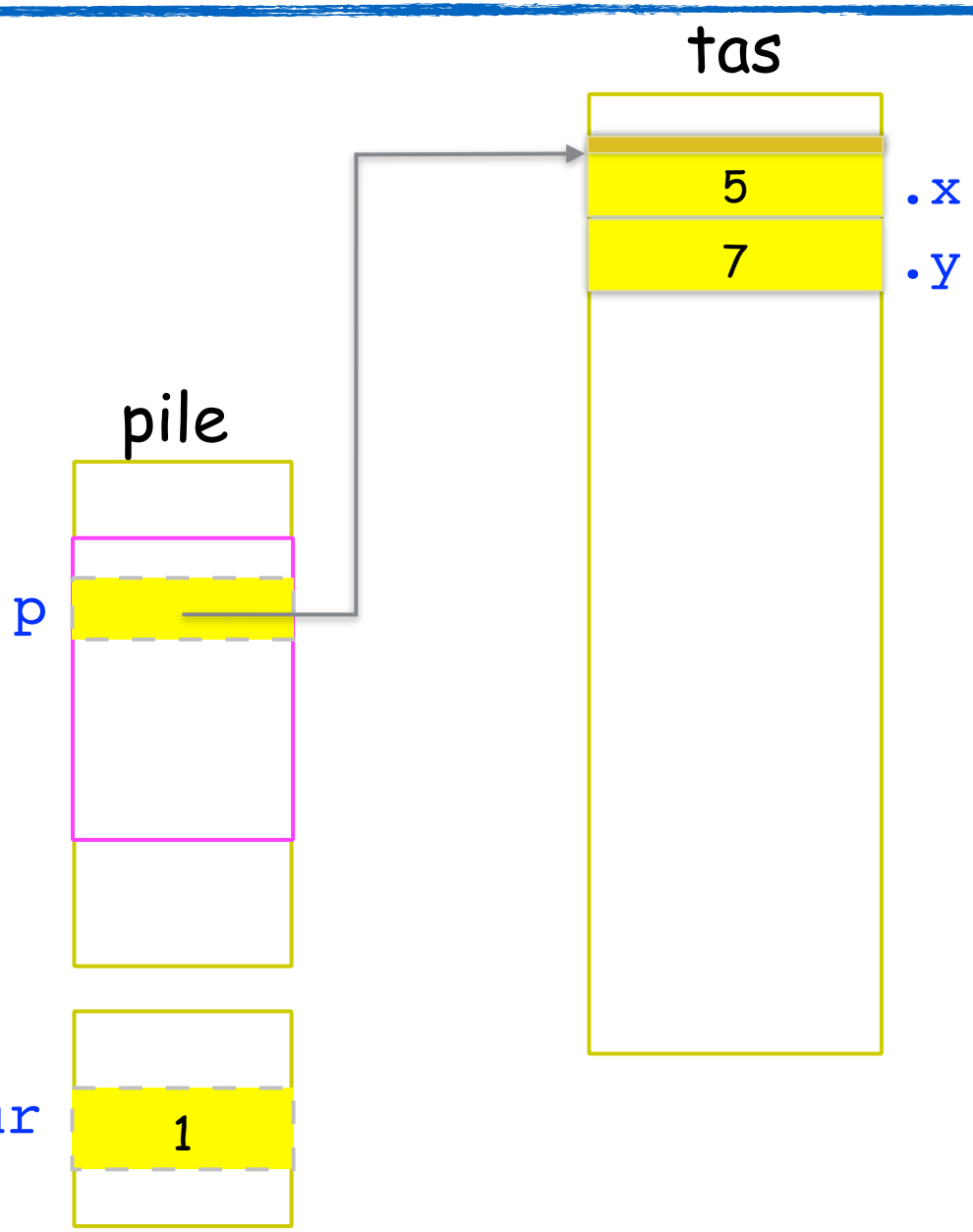
```
class Point{
  private
   double x, y;
  private static
   int compteur=0;
  …
};
```

Chargement de la classe



Champs "static" (variables de classe)

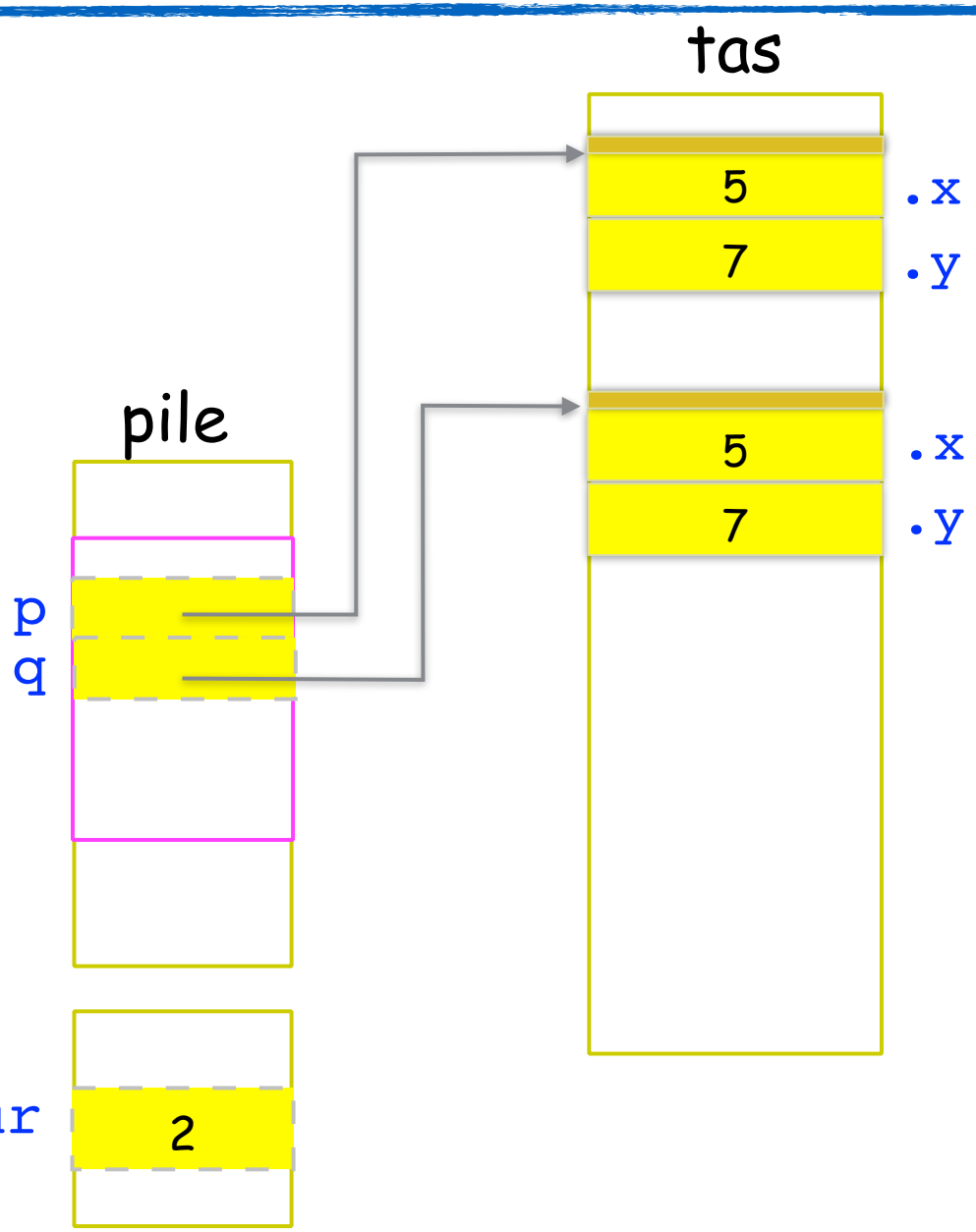
```
class Point{
  private
   double x, y;
  private static
   int compteur=0;
  …
};
…
Point p =
new Point (5,7);
```



Execution

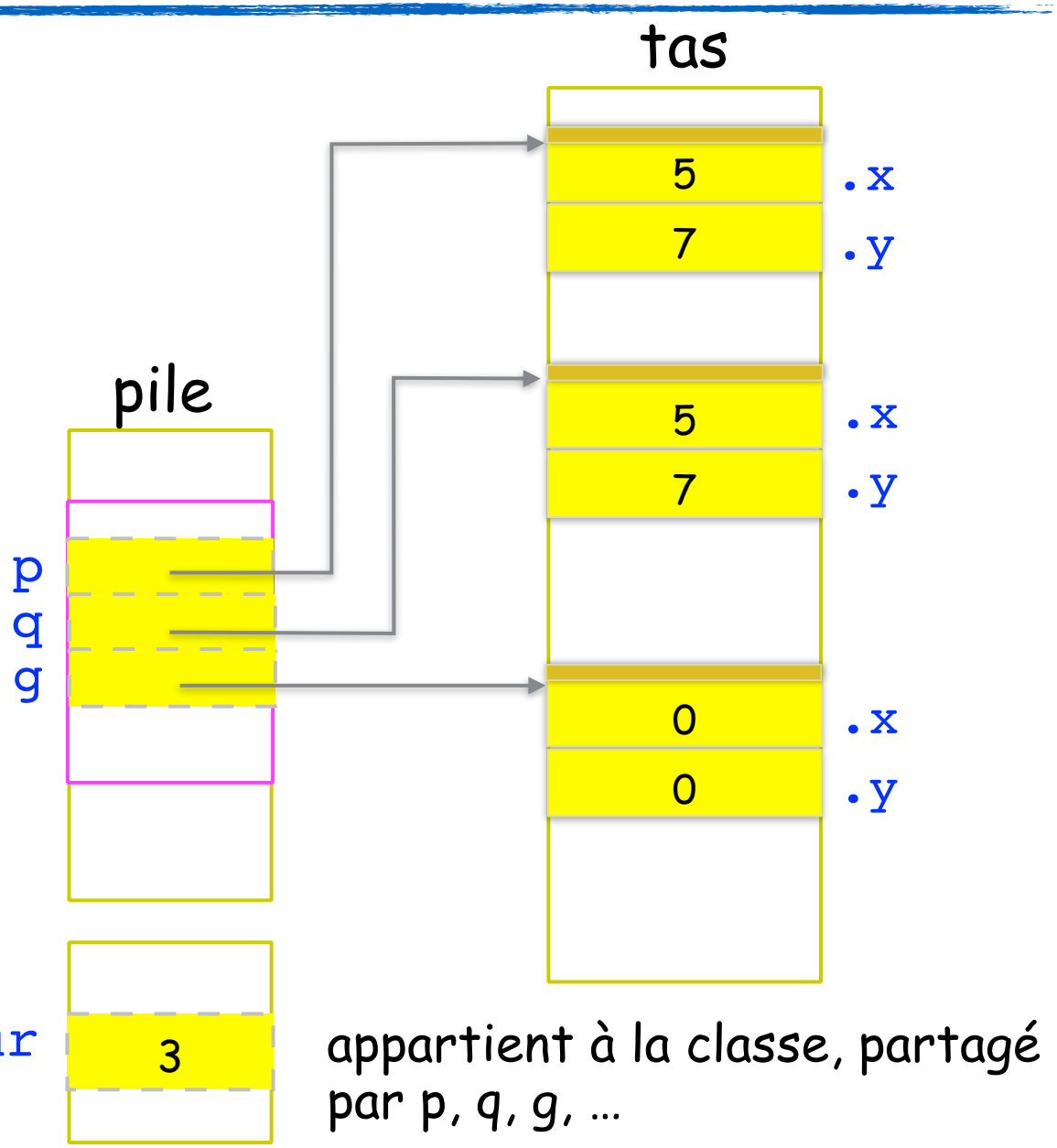
Champs "static" (variables de classe)

```
class Point{
  private
   double x, y;
  private static
    int compteur=0;
  …
};
…
Point p =
new Point (5,7);
Point q =
new Point (5,7);
```



Champs "static" (variables de classe)

```
class Point{
  private
   double x, y;
  private static
    int compteur=0;
  …
};
…
Point p =
new Point (5,7);
Point q =
new Point (5,7);
Point g =
new Point (0,0);
```



appartient à la classe, partagé par p, q, g, …

Champs "static" (variables **de classe**)

- Pas besoin d'un objet pour accéder à un membre static
- Exemple : la classe Math de java.lang :

```
public class Math {
  public static final double PI =  3,141592…
  …
}
```

- Accès : `NomClasse.nomMembre`

```
double c = 2 * Math.PI
```

Méthodes "static" (méthodes de classe)

- De façon similaire, une méthode static appartient à la classe, pas besoin d'un objet pour l'invoquer

```
class Point {
  private double x, y;
  private static int compteur = 0;
  public Point(double pX, double pY)
  { x = pX; y = pY; compteur++;}
  // methodes
  public static double nombrePoints()
  { return compteur;}
  …
}
class Test {
   public static void main (String[] args) {
    System.out.println
       (Point.nombrePoints()); //0
   }
}
```

Méthodes "static" (méthodes de classe)

- Mais les membres static peuvent également être invoqués par des objets

```
class Test {
   public static void main (String[] args) {
    System.out.println (Point.nombrePoints()); //0
    Point p = new Point(2,3);
    System.out.println (p.nombrePoints()); //1
   }
}
```

- L'accès par la classe est préférable (code plus clair)

Méthodes "static" (méthodes de classe)

- Remarque : le main est toujours une méthode static

```
class Test {
   public static void main (String[] args) {
     …
   }
}
```

- Cela permet son invocation automatique à l'exécution, sans création d'un objet de la classe Test

Méthodes "static" (méthodes de classe)

- Une méthode static peut accéder uniquement à des membres static de sa classe, et n'a pas d'accès à `this`
  - motivation : une méthode static n'est attachée à aucun objet (e.g. peut être invoquée sans qu'aucun objet soit crée)
    - si elle accédait à un champ d'instance : de quel objet?

```
class Point {
  private double x, y;
  private static int compteur = 0;
  public static double nombrePoints() {
   x = 5; //ERREUR
   this.deplace(1,2); //ERREUR
   return compteur;
  }
  …
 }
…
 Point.nombrePoints();
```

Méthodes de classe / méthodes d'instance

- Une méthode d'instance peut accéder à tous les membres de sa classe (static ou non)

```
class Point {
  private double x, y;
  private static int compteur = 0;
  public Point(double pX, double pY)
  { x = pX; y = pY; compteur++;}
  public static double nombrePoints()
  { return compteur;}
  …
}
```

Constructeurs

- Appelés par l'opérateur `new` pour créer un objet
- Initialisent les objets; peuvent avoir des paramètres
- Plusieurs constructeurs possibles (avec surcharge, voir plus loin)
  - à condition que les types des paramètres soient different

```
class Point {
  private double x, y;
  private static int compteur = 0;
  public Point(double pX, double pY)
  { x = pX; y = pY; compteur++;}
  public Point ()
  { x = 0; y = 0; compteur++;}
  //methodes
  …
 }
```

Constructeurs

- Constructeur par défaut (si aucun constructeur n'est défini)

  `public NomClasse(){}`

```
class A {
  private int x, y;
}
…
A a = new A(); //permis
```

```
class A {
  private int x, y;
  public A (int x, int y){…}
}
…
A a = new A(); //ERREUR
```

Constructeurs

- `this()` pour appeler un constructeur dans un autre constructeur
  - si présent doit être la première instruction

```
class Point {
  private double x, y;
  private static int compteur = 0;
  public Point(double pX, double pY){
   x = pX; y = pY; compteur++;
  }
  public Point () {
   this (0,0);
  }
  public Point ( Point p) {
   this (p.x, p.y);
  }
  //methodes
  …
}
```

copie par constructeur

Modificateurs de classes

- Plusieurs modificateurs peuvent précéder la définition d'une classe
- Un modificateur peut être :
  - modificateur d'acces
    - `public` : classe visible à toute autre classe
    - pas de modificateur : visibilité package
  - `abstract` (incomplète, pas d'instance)
  - `final` (pas d'extension)
  - `strictfp` (pour les « réels »)

- La définition d'une classe peut être aussi précédée par une annotation
  - meta-information sur la classe

Modificateurs de membre

Pour tous les membres (rappel : les constructeurs et les blocs d'initialisation ne sont pas des membres)

- contrôle d'accès
  - `private` (visible uniquement par la classe)
  - `public` (visible par tout le monde)
  - pas de modificateur (package) (visible uniquement par les classes du même package)
  - `protected` (visible uniquement par les classes du même package et par les sous-classes)
- static (membre commun à la classe)
- final (champ constant / classe non extensible / méthode non-redefinissable)
- annotations

D'autres modificateurs de membre

Pour les champs

- transient, volatile

Pour les méthodes

- abstract, synchronized, native, strictfp

Pour le classes membres

- abstract, strictfp

Modificateurs : exemple

```
class Point {
  private double x, y;
  …
  public void deplace(double newX, double newY) {…}
  public double getX(){…}
  …
}
class Geometry{
   public static void main (String [] args) {
      Point p = new Point(5,7);
      p.x = 0; //ERREUR
      p.deplace (0,7); //OK
      System.out.println (p.x); //ERREUR
      System.out.println (p.getX()); //OK
   }
}
```

Modificateurs : remarques

- Tous les champs (même privés) sont accessibles à l'intérieur de la classe
  - sur `this` comme sur tout autre objet de la classe !

```
class Point {
  private double x, y;
  …
  public Point(double pX, double pY){
    x = pX; y = pY; compteur++;
  }
  public Point ( Point p) {
    this (p.x, p.y);
  }
}
```

Passage des paramètres aux méthodes

- Le passage des paramètres en Java est toujours par valeur
  - i.e. la valeur du paramètre passé est copiée dans la mémoire locale de la méthode (RA)
- Paramètres de type primitif / type référence : effet différent

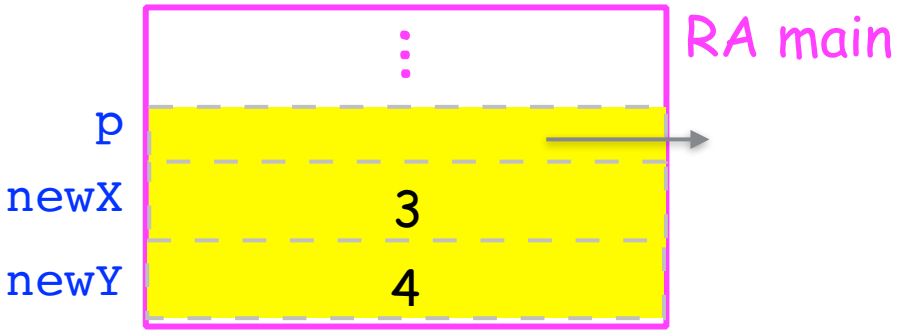
```
Class Point {
 …
 public void deplace (double newX, double newY)
 { … }
 public double distance( Point p ){
    return Math.sqrt((p.x-x)*(p.x-x)+(p.y-y)*(p.y-y));
 }

 …
}
```

Paramètres de type primitif : exemple

```
Class Point {
 public void deplace (double newX, double newY) { …
 … }
}
```

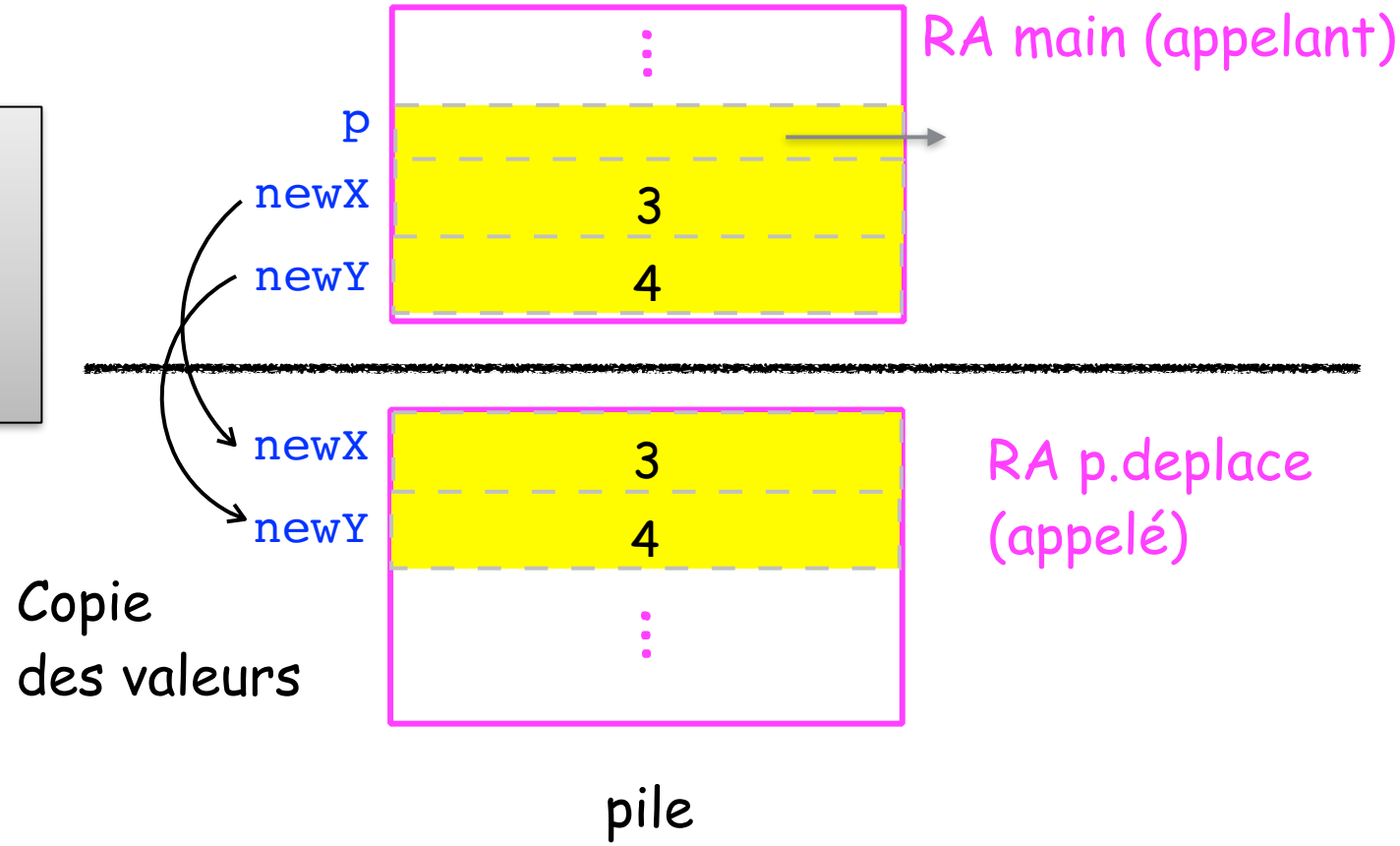
```
//main
Point p = …;
newX=3; newY=4;
```



Paramètres de type primitif : exemple

```
Class Point {
 public void deplace (double newX, double newY) { …
 … }
}
```

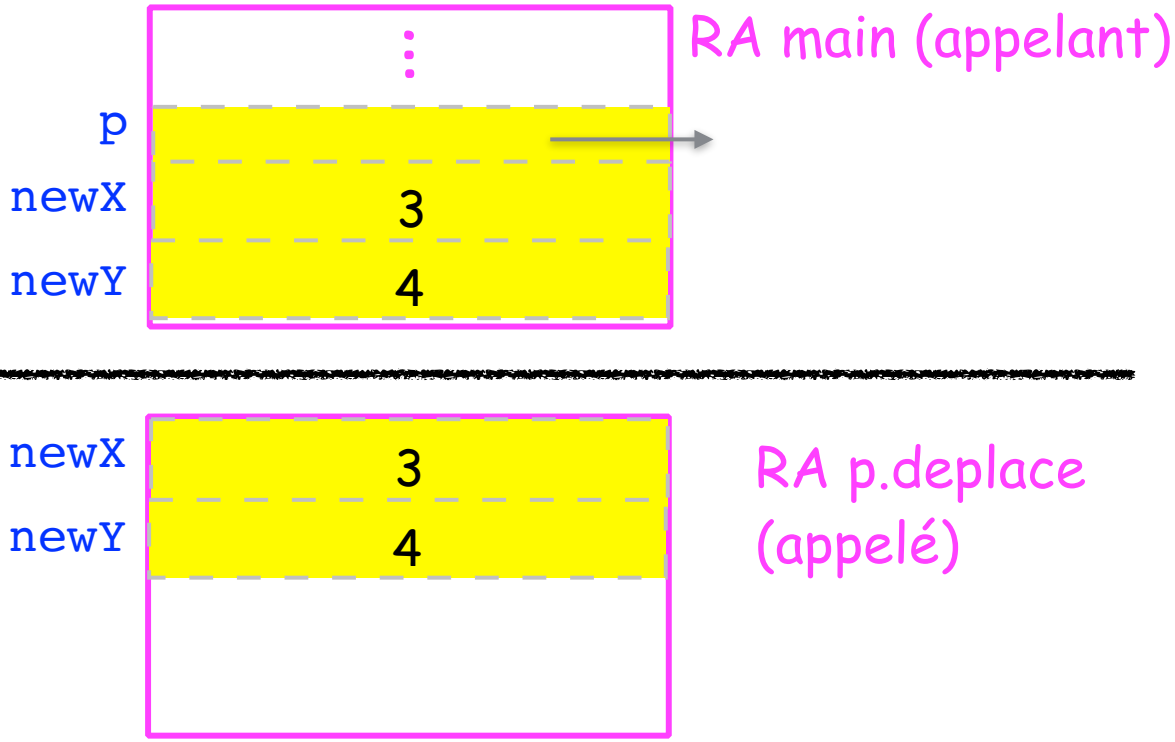
```
//main
Point p = …;
newX=3; newY=4;
p.deplace
    (newX,newY);
```



Paramètres de type primitif : exemple

```
Class Point {
 public void deplace (double newX, double newY) { …
 … newX=0; newY=0; }
}
```

```
//main
Point p = …;
newX=3; newY=4;
p.deplace
   (newX,newY);
```

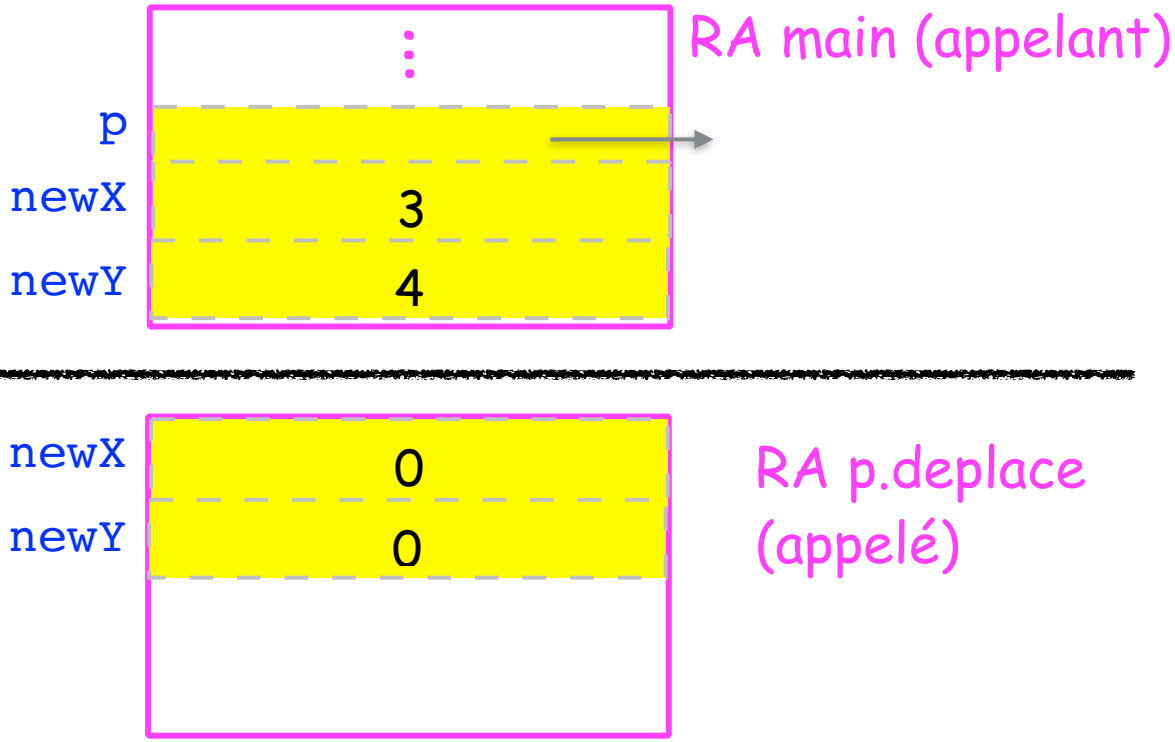


- Toute modification des paramètres par p.deplace n'affecte pas les variables du main

Paramètres de type primitif : exemple

```
Class Point {
 public void deplace (double newX, double newY) { …
 … newX=0; newY=0; }
}
```

```
//main
Point p = …;
newX=3; newY=4;
p.deplace
   (newX,newY);
```

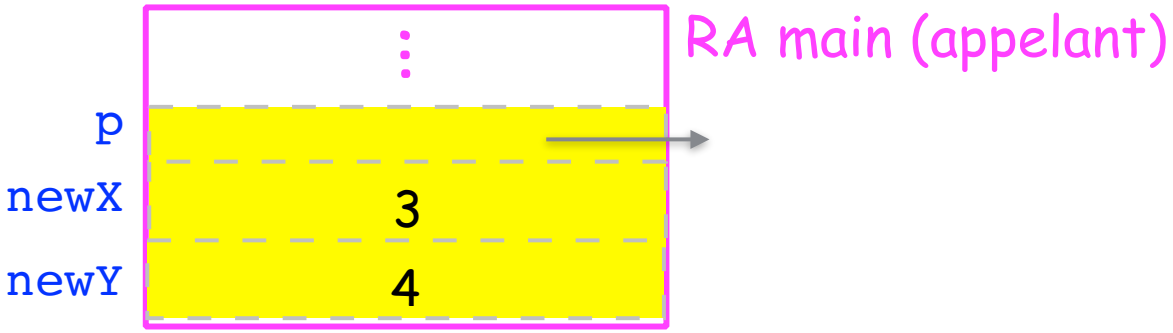


- Toute modification des paramètres par p.deplace n'affecte pas les variables du main

Paramètres de type primitif : exemple

```
Class Point {
 public void deplace (double newX, double newY) { …
 … newX=0; newY=0; }
}
```

```
//main
Point p = …;
newX=3; newY=4;
p.deplace
   (newX,newY);
System.out.print
(newX);//3
```

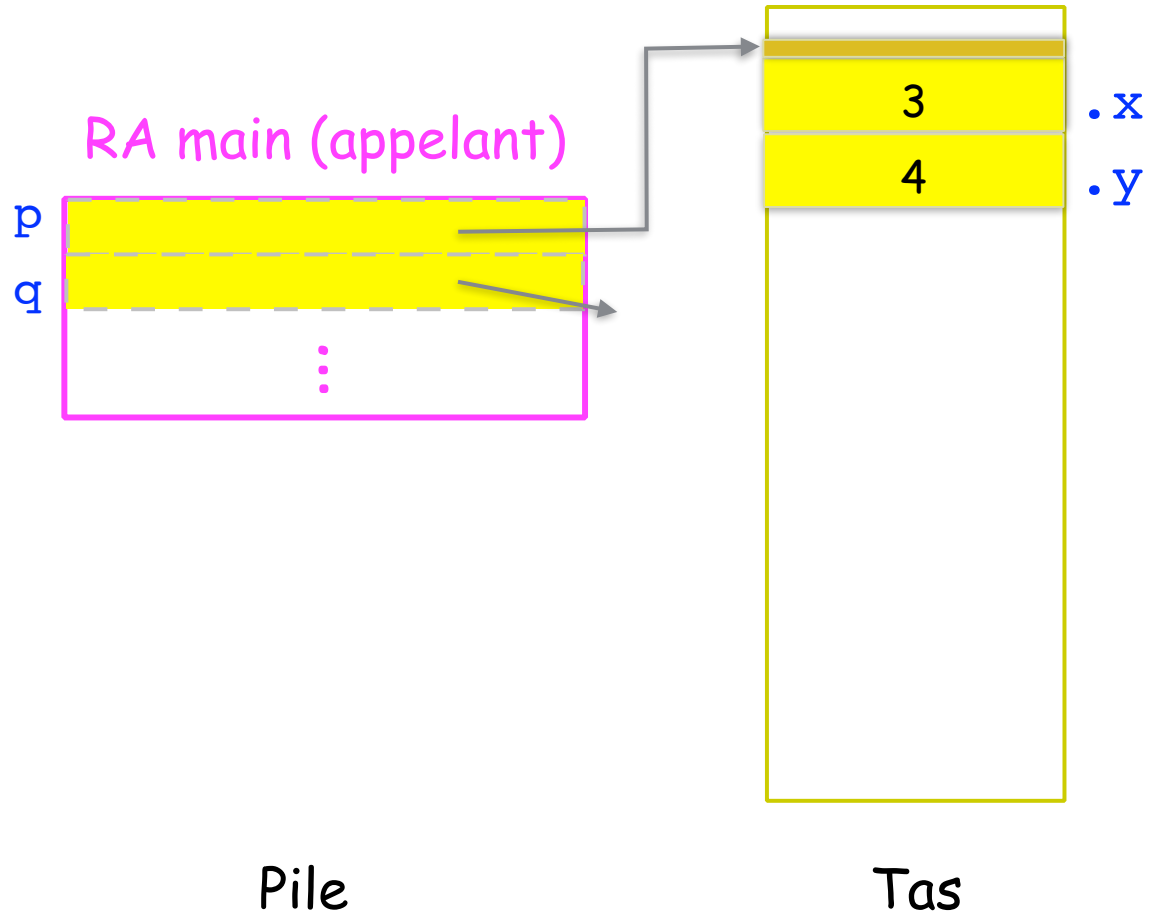


- Toute modification des paramètres par p.deplace n'affecte pas les variables du main

Paramètres de type référence : exemple

```
Class Point {
 public double distance (Point p) { …
 … }
}
```

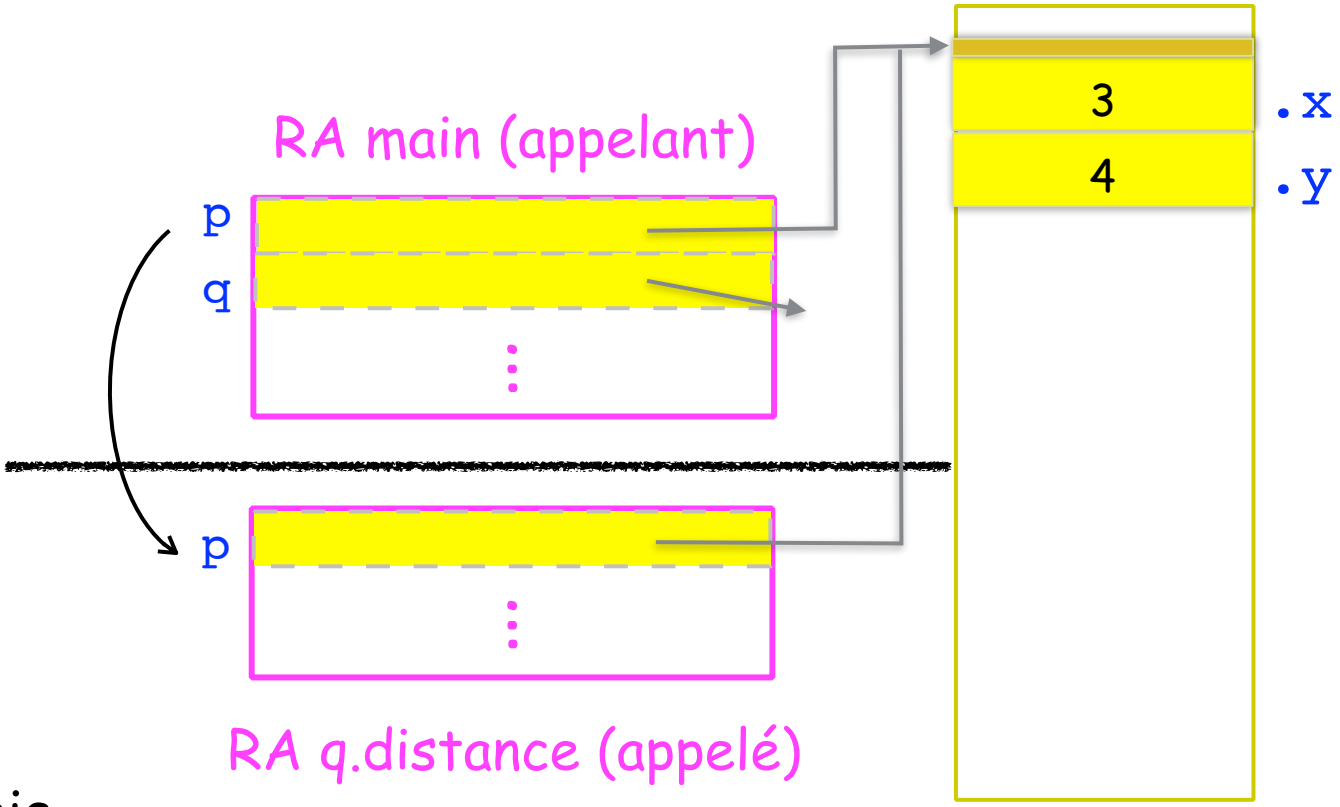
```
//main
Point p = new
    Point(3,4);
Point q =…;
```



Paramètres de type référence : exemple

```
Class Point {
 public double distance (Point p) { …
 … }
}
```

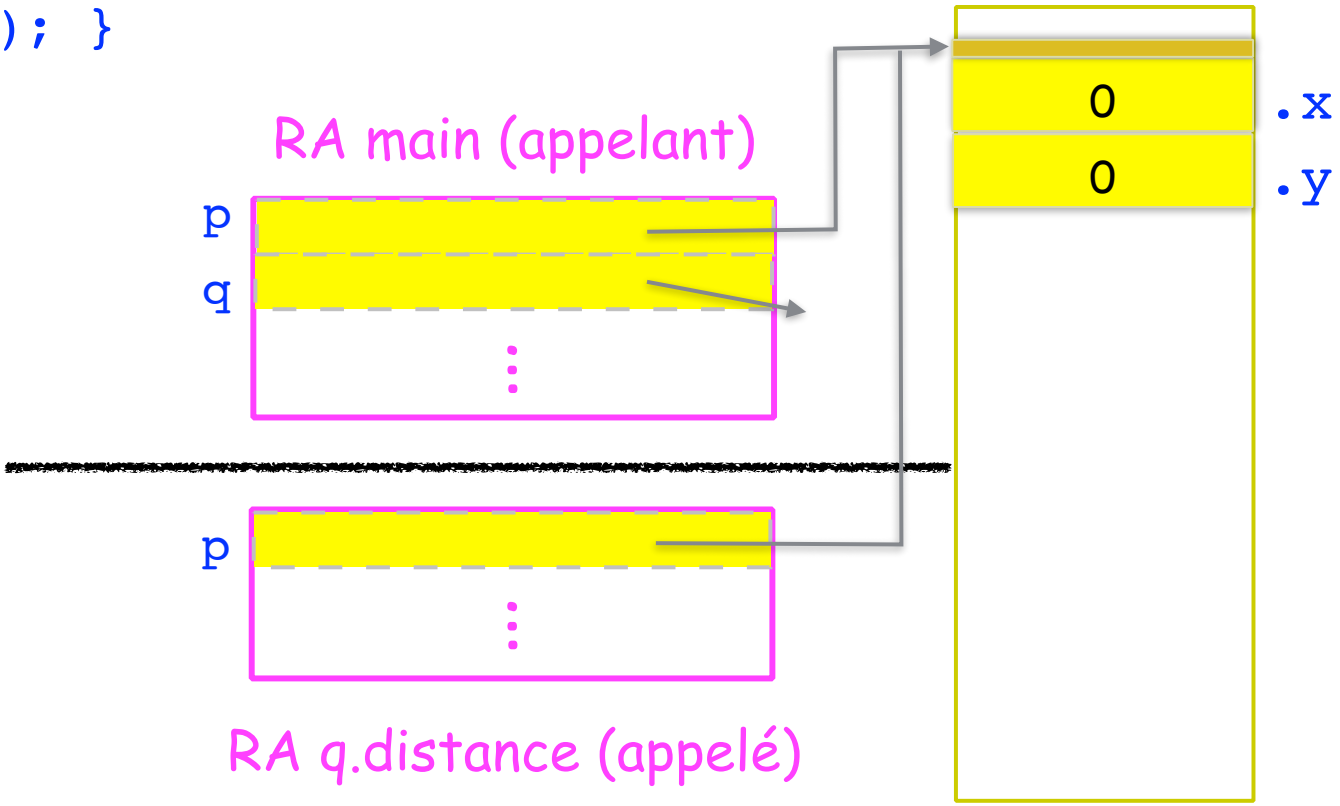
```
//main
Point p = new
    Point(3,4);
Point q =…;
System.out.print
(q.distance(p));
```



Paramètres de type référence : exemple

```
Class Point {
 public double distance (Point p) { …
 … p.deplace(0,0); }
}
```

```
//main
Point p = new
   Point(3,4);
Point q =…;
System.out.print
(q.distance(p));
```

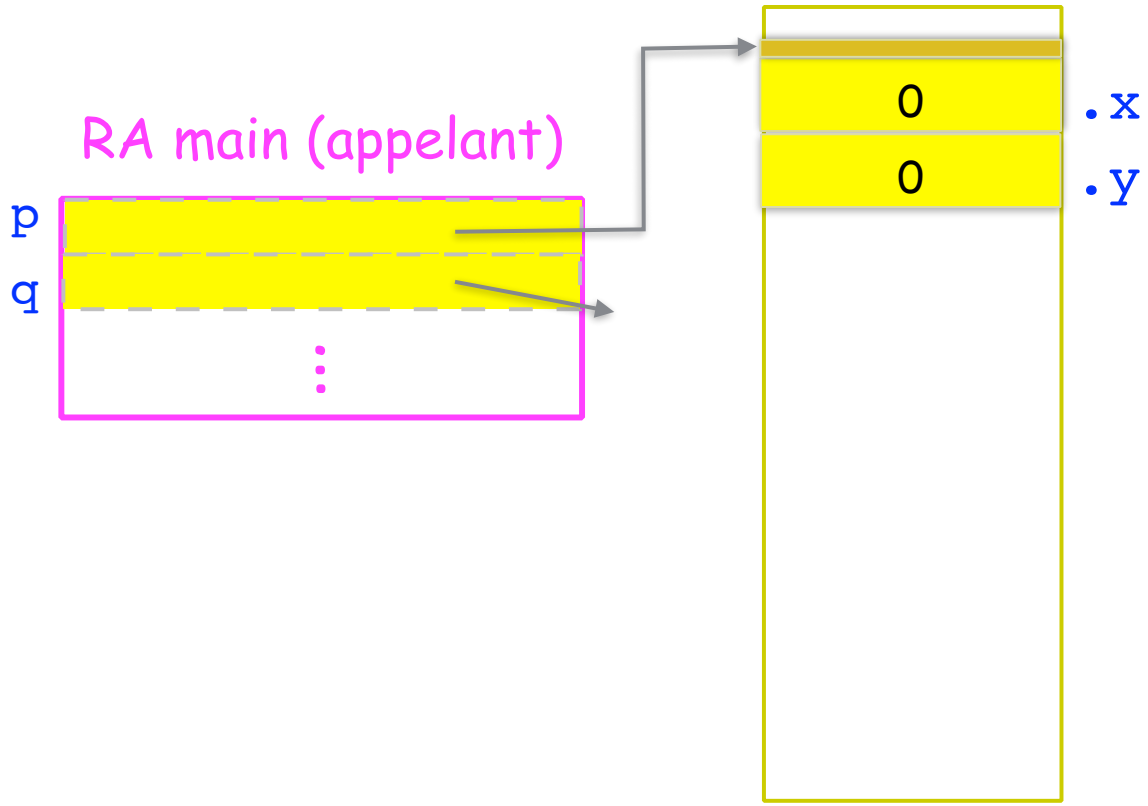


- Toute **modification de l'objet référencé par p** faite par q.distance **affecte le main** (puisqu'il partage l'objet)

Paramètres de type référence : exemple

```
Class Point {
 public double distance (Point p) { …
 … p.deplace(0,0); }
}
```

```
//main
Point p = new
   Point(3,4);
Point q =…;
System.out.print
(q.distance(p));
System.out.print
(p.getX());//0
```

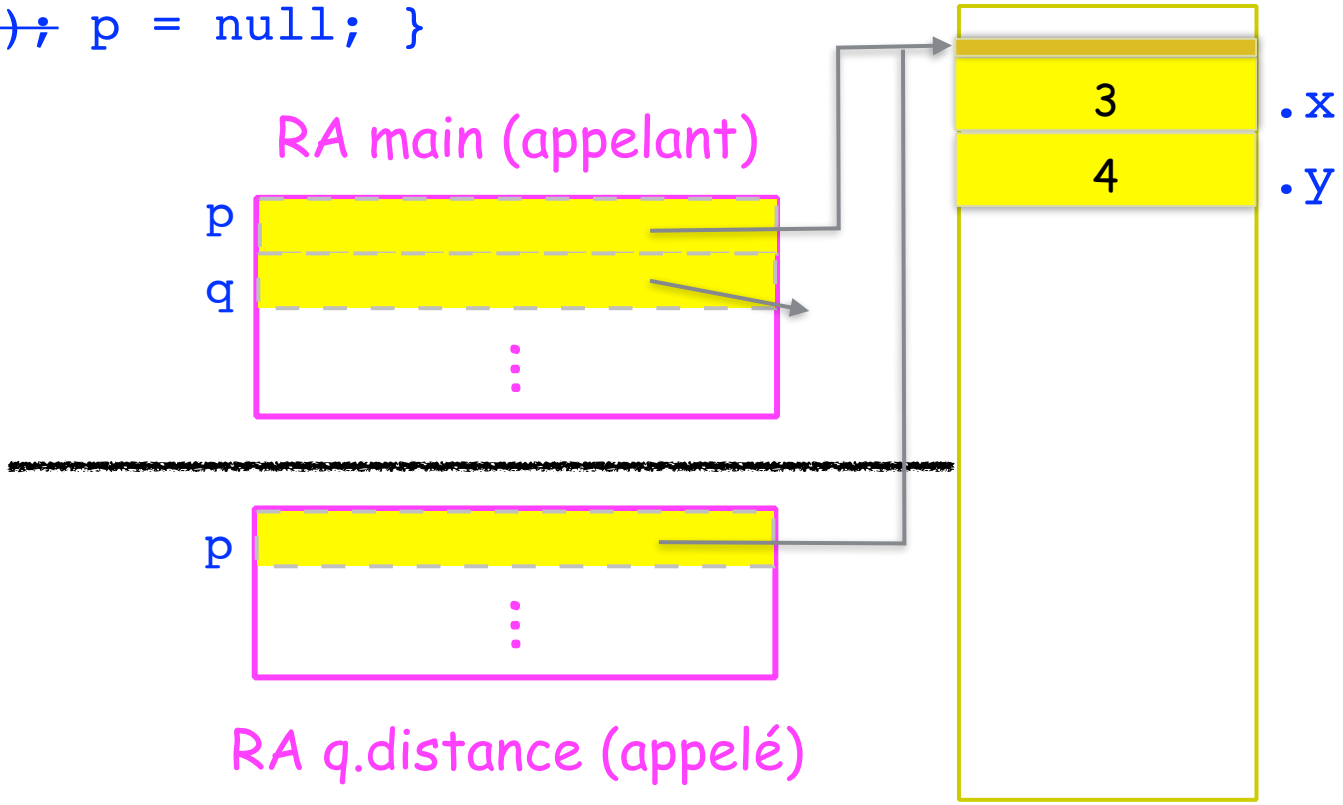


- Toute **modification de l'objet référencé par p** faite par q.distance **affecte le main** (puisqu'il partage l'objet)

Paramètres de type référence : exemple

```
Class Point {
 public double distance (Point p) { …
 … p.deplace(0,0); p = null; }
}
```

```
//main
Point p = new
    Point(3,4);
Point q =…;
System.out.print
(q.distance(p));
```

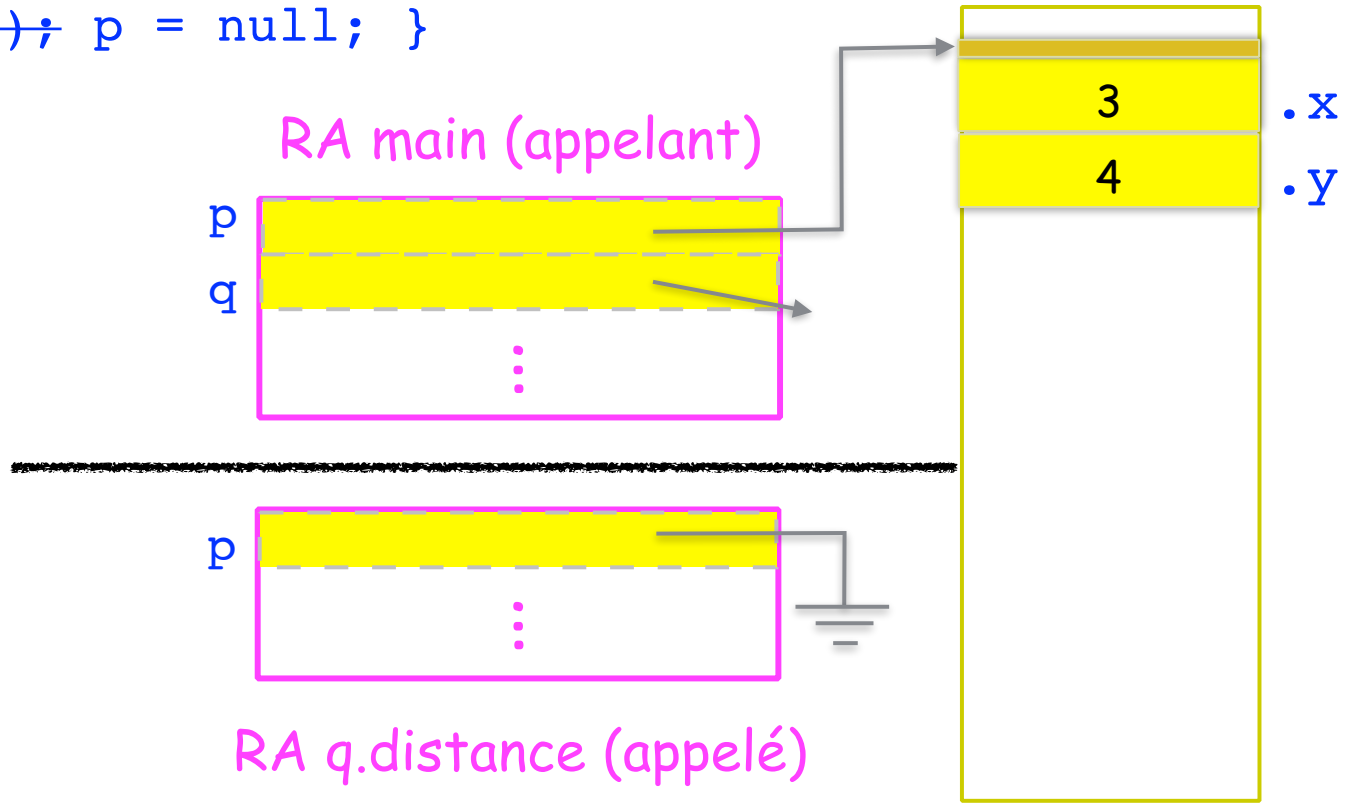


- Toute **modification de la référence p** par q.distance **n'affecte pas le main**

Paramètres de type référence : exemple

```
Class Point {
 public double distance (Point p) { …
 … p.deplace(0,0); p = null; }
}
```

```
//main
Point p = new
    Point(3,4);
Point q =…;
System.out.print
(q.distance(p));
```



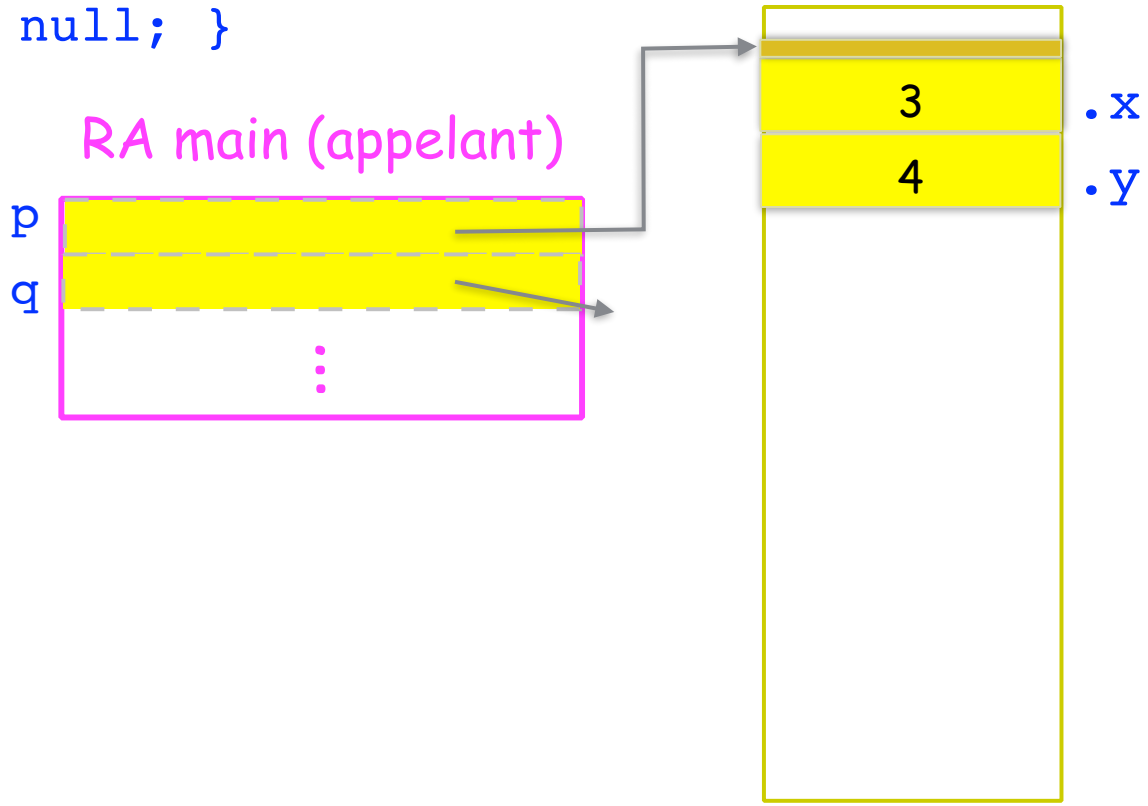
- Toute **modification de la référence p** par q.distance **n'affecte pas le main**

Paramètres de type référence : exemple

```
Class Point {
 public double distance (Point p) { …
 … p.deplace(0,0); p = null; }
}
```

```
//main
Point p = new
   Point(3,4);
Point q =…;
System.out.print
(q.distance(p));

System.out.print
(p.getX());//3
```



- Toute **modification de la référence p** par q.distance **n'affecte pas le main**