

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 familles 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 classe
 - 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

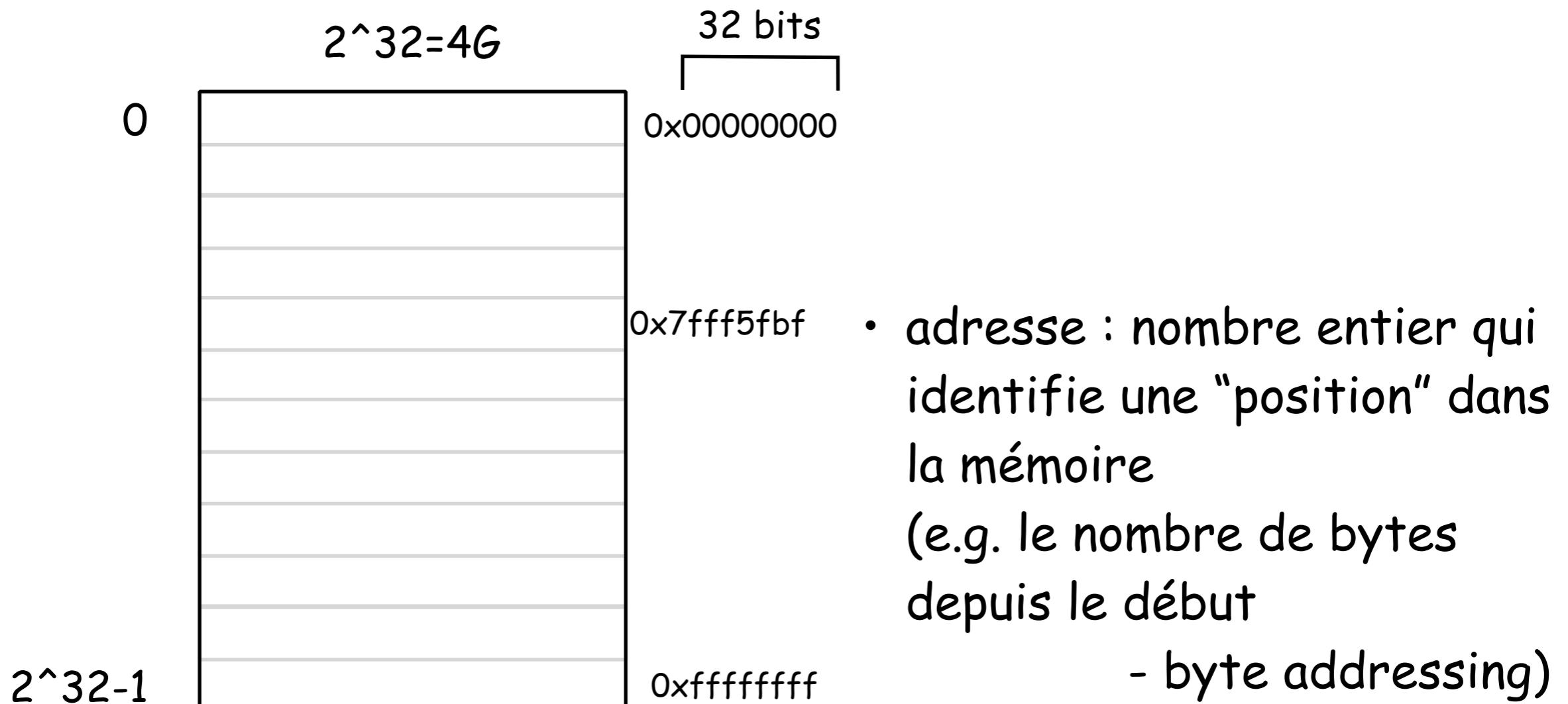
```
class Point { ... }
public class Geometry {
    public static void main( String[ ] args ){
        Point p = new Point(5,7);
        // p fait référence à un objet de classe Point
        // p.x vaut 5 , p.y vaut 7
        Point q = new Point(5,7);
        // q fait référence à un autre objet de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

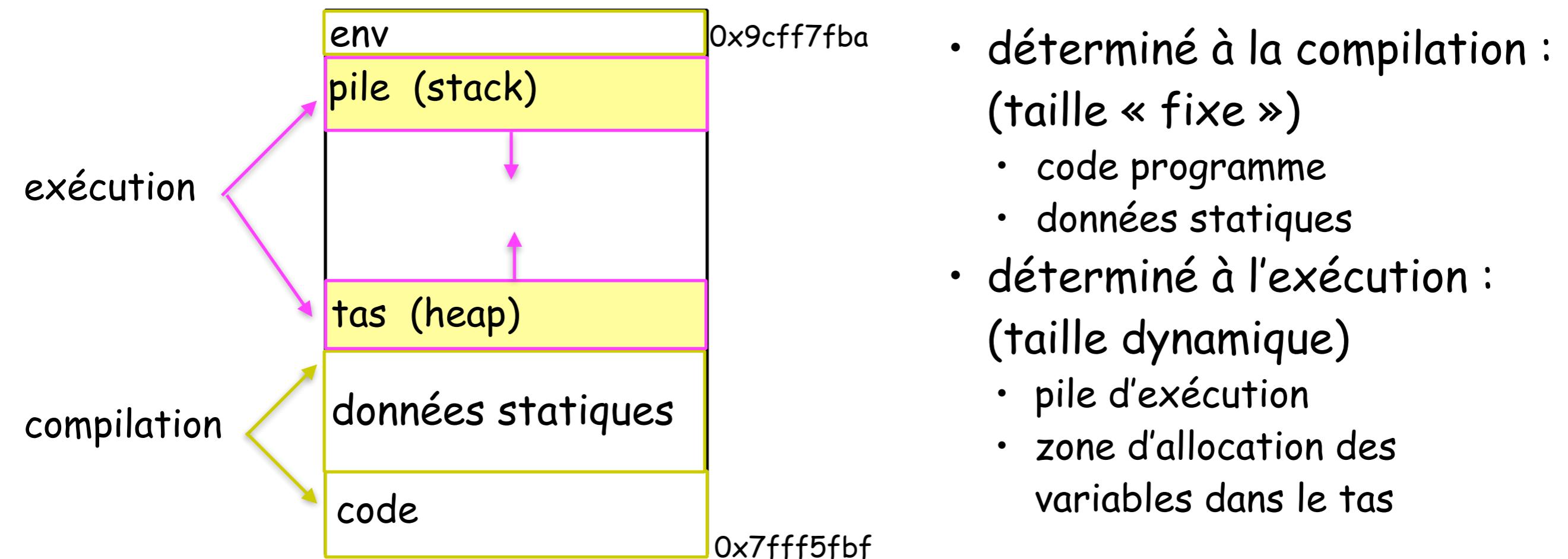
Espace mémoire

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



La mémoire d'un programme

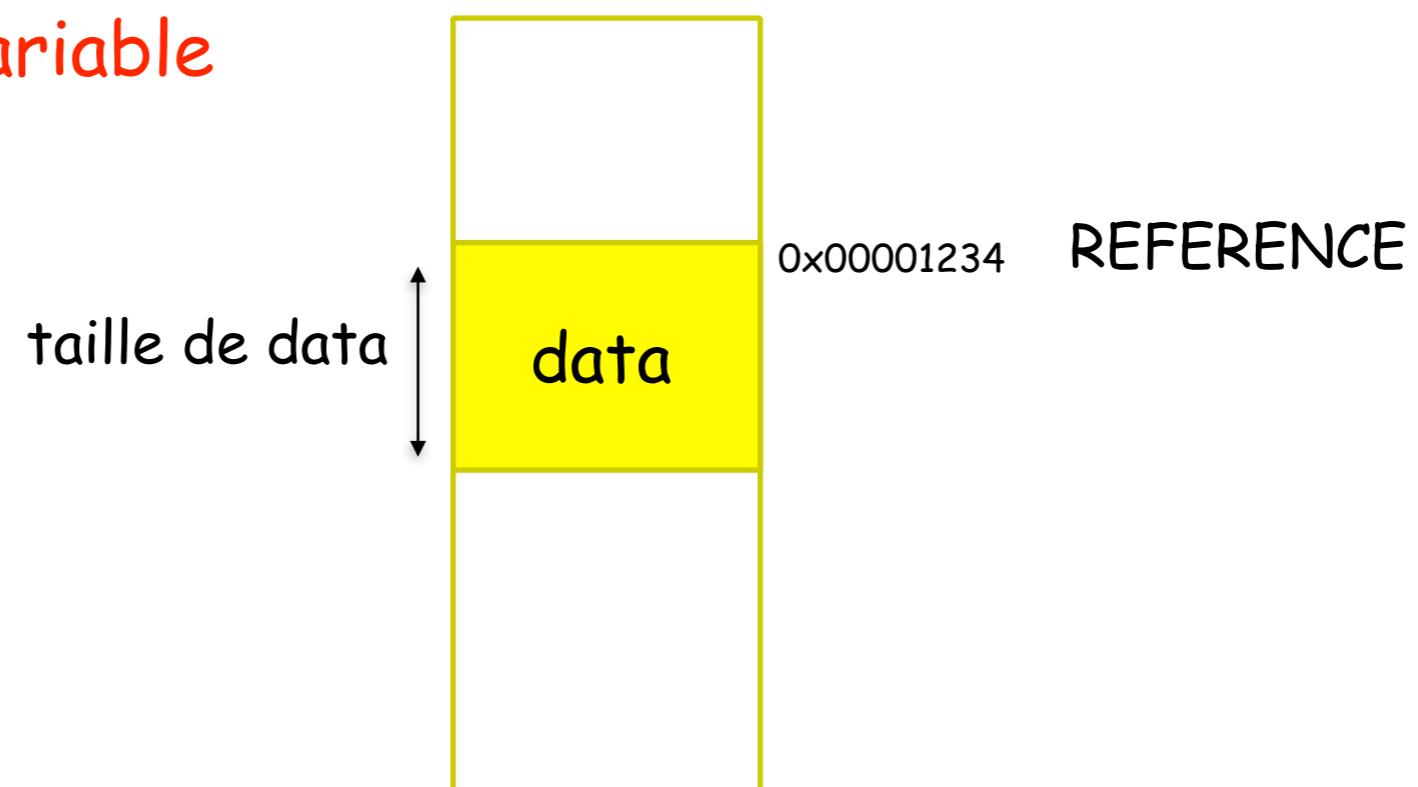
- Mémoire allouée pour un programme (modèle similaire pour la machine virtuelle)



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

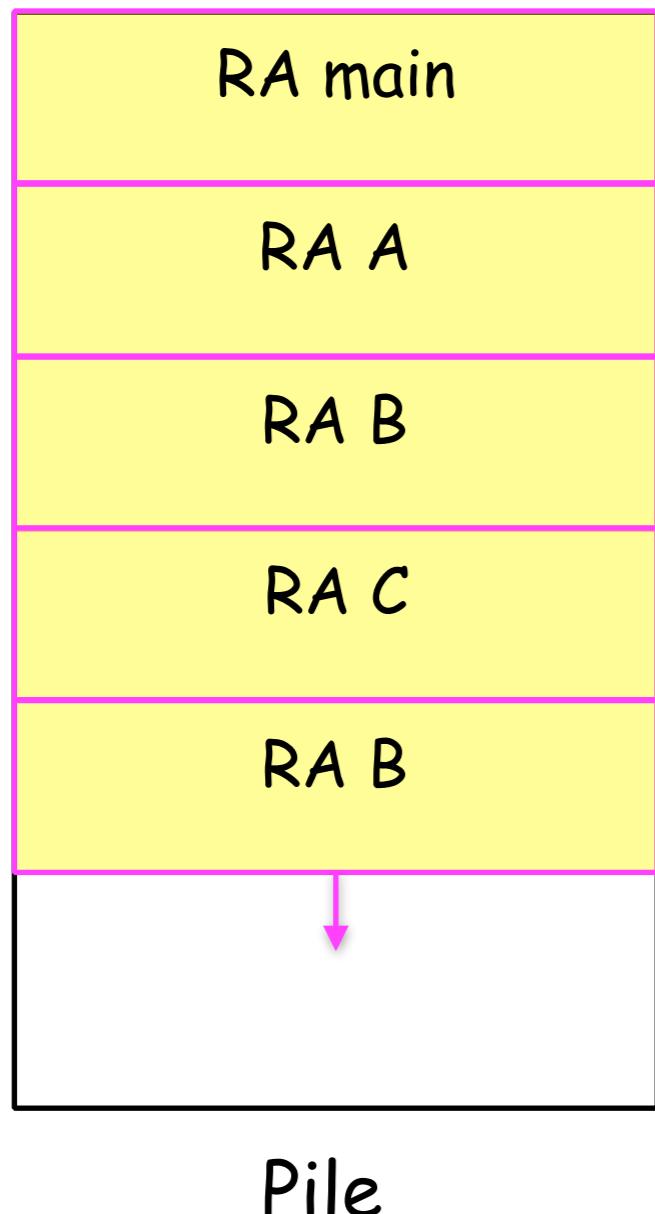
Allocation d'une variable



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

La pile d'exécution

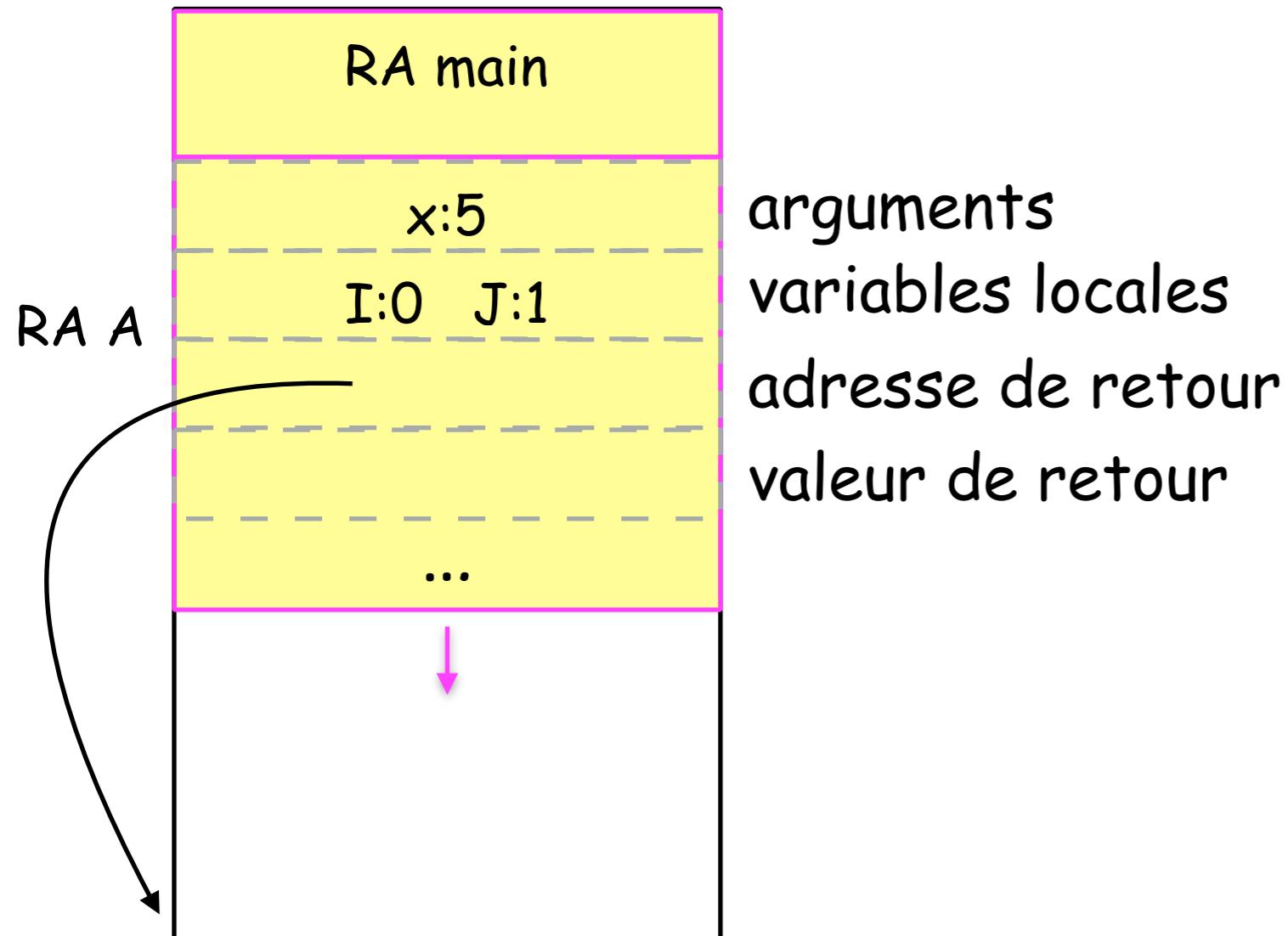
- Quand une fonction (méthode) est appelée, un nouvel espace (dit **record d'activation ou <<frame>>**) est empilé sur la pile d'exécution
- Ensuite compteur de programme (**PC**) -> l'adresse de la fonction



```
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
```

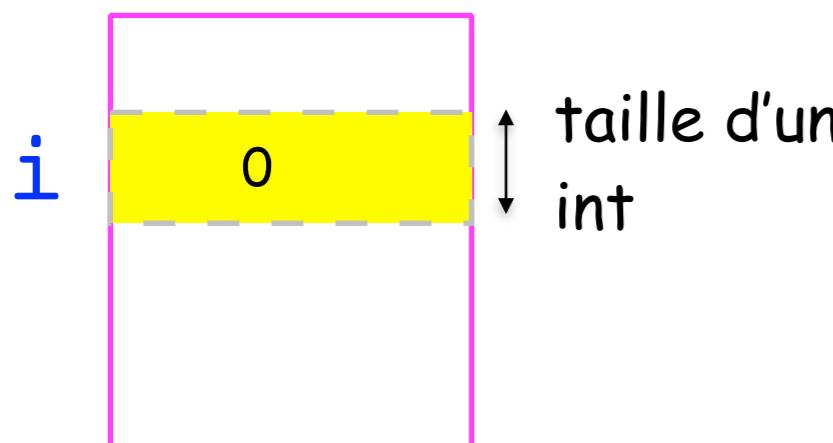
- **Adresse de retour** : adresse de l'instruction à exécuter quand la fonction termine
- Quand la fonction **A()** termine, **RA A** est dépiler

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

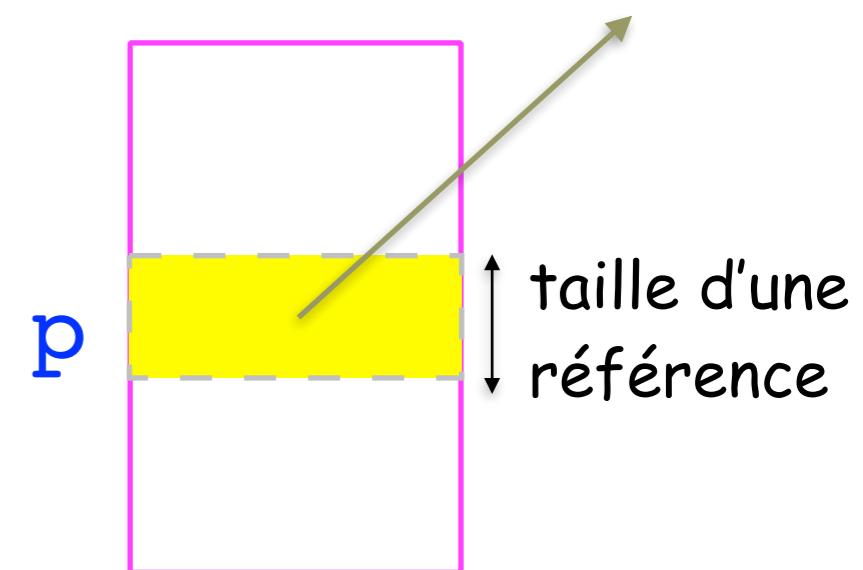
```
int i=0;
```



RA dans la pile

Objet

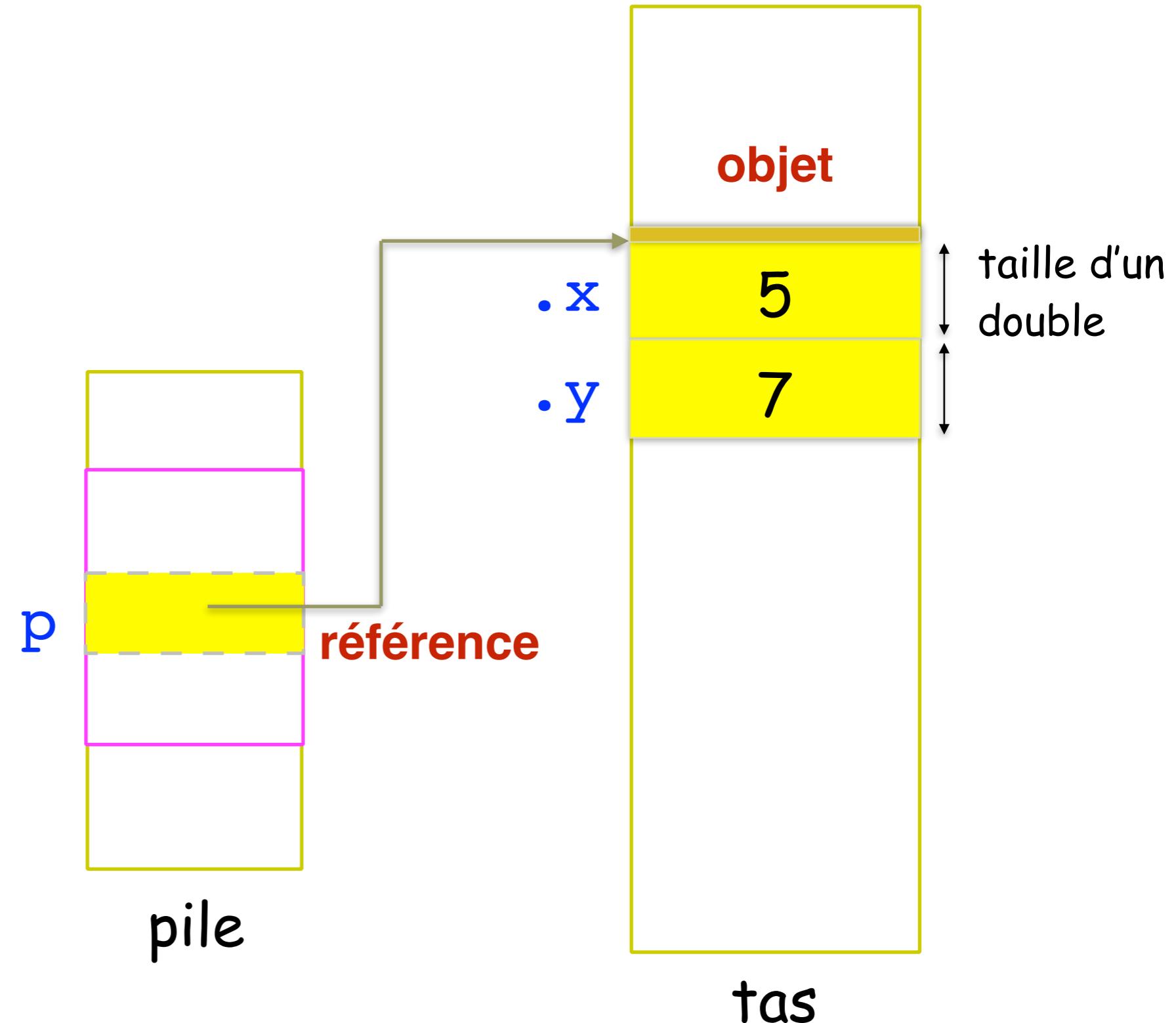
```
class Point{  
    private  
    double x, y;  
  
    ...  
};  
...  
Point p =  
new Point (5,7);
```



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);

        ...
    }
}
```

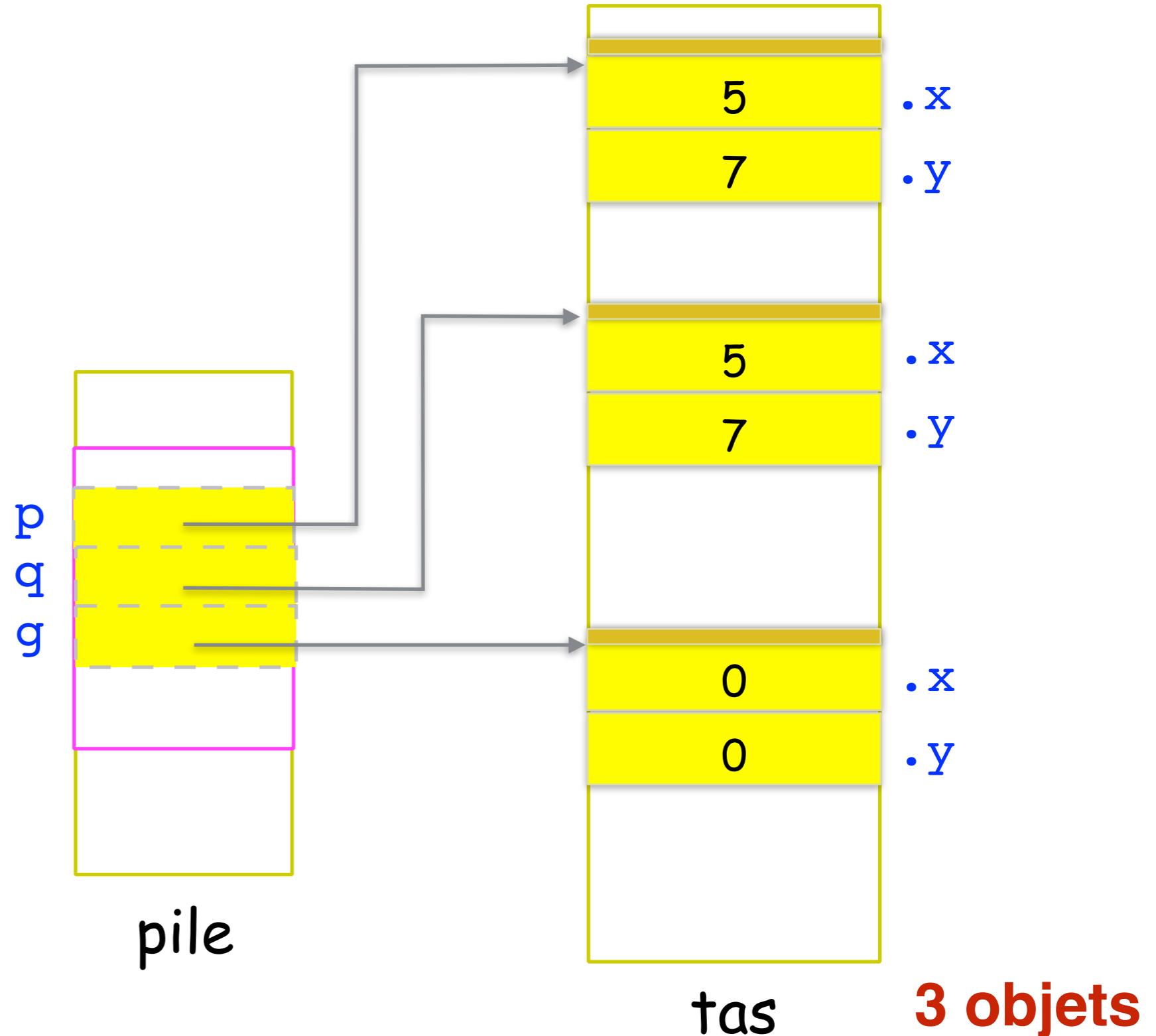
Variables d'instance

- 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`

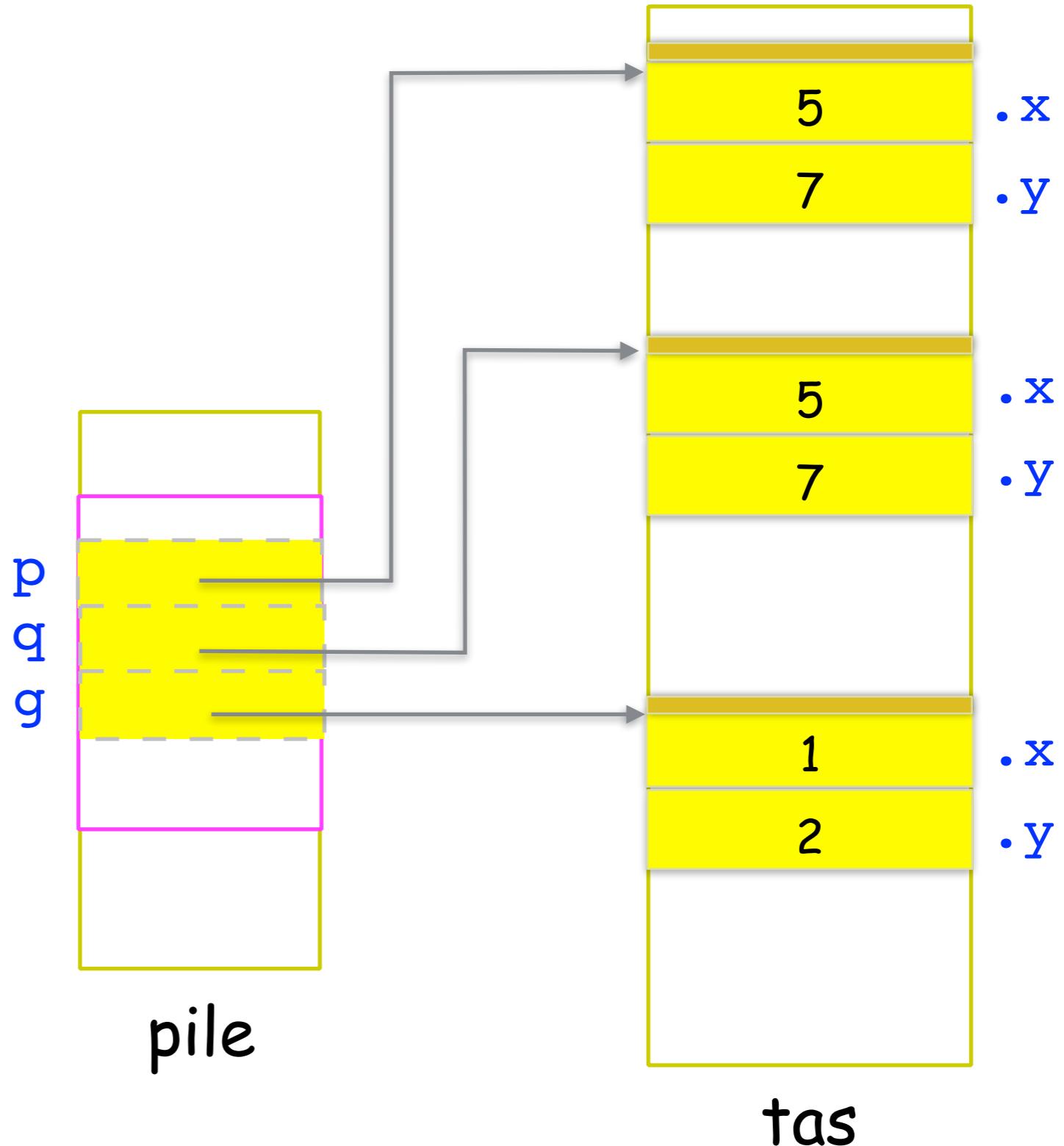
```
Class Point {  
    ...  
    public void deplace (double newX, double newY)  
    { this.x = newX; y = newY; }  
    ...  
}
```

- `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)

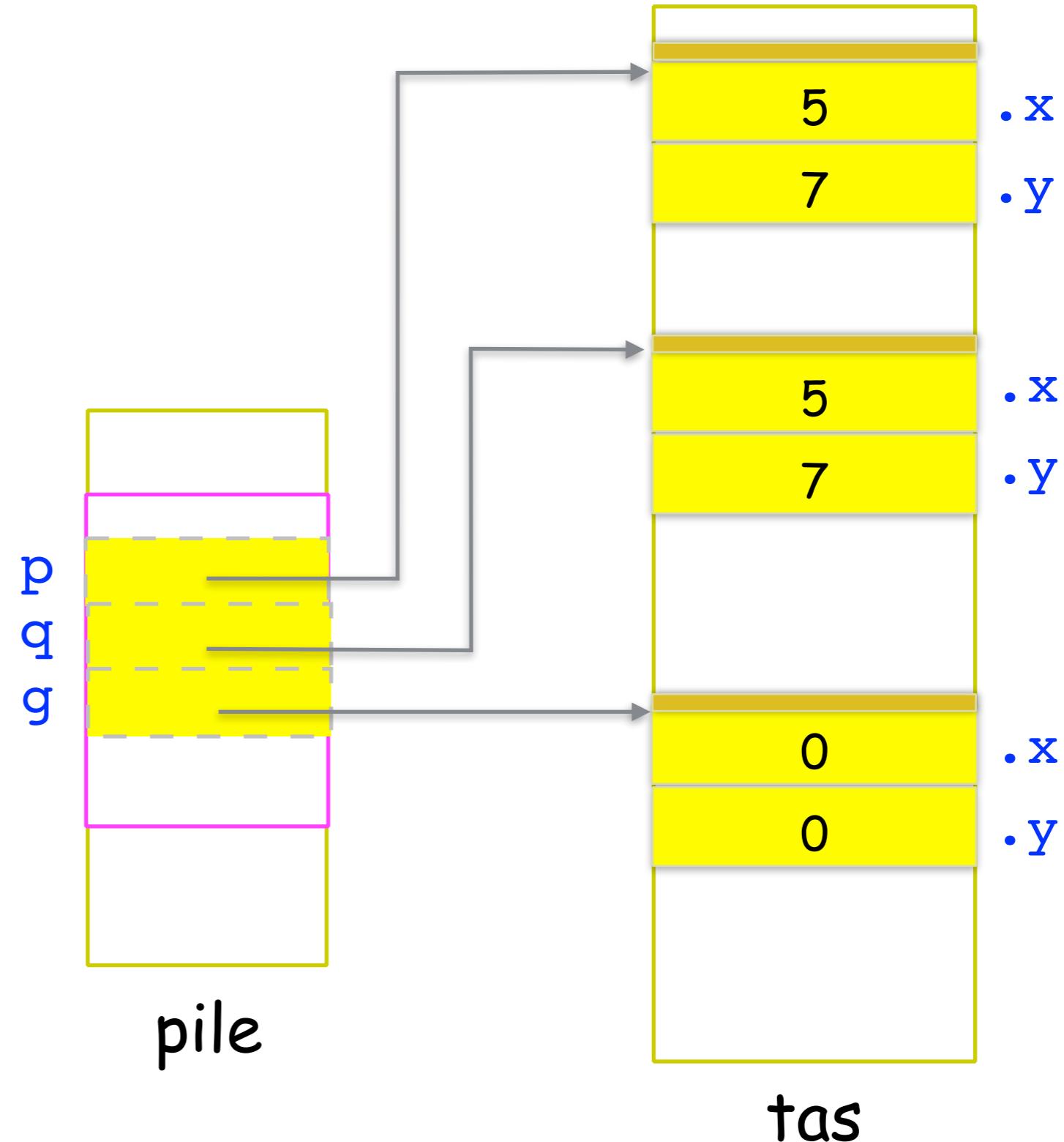


Références et objets : test d'égalité

```
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

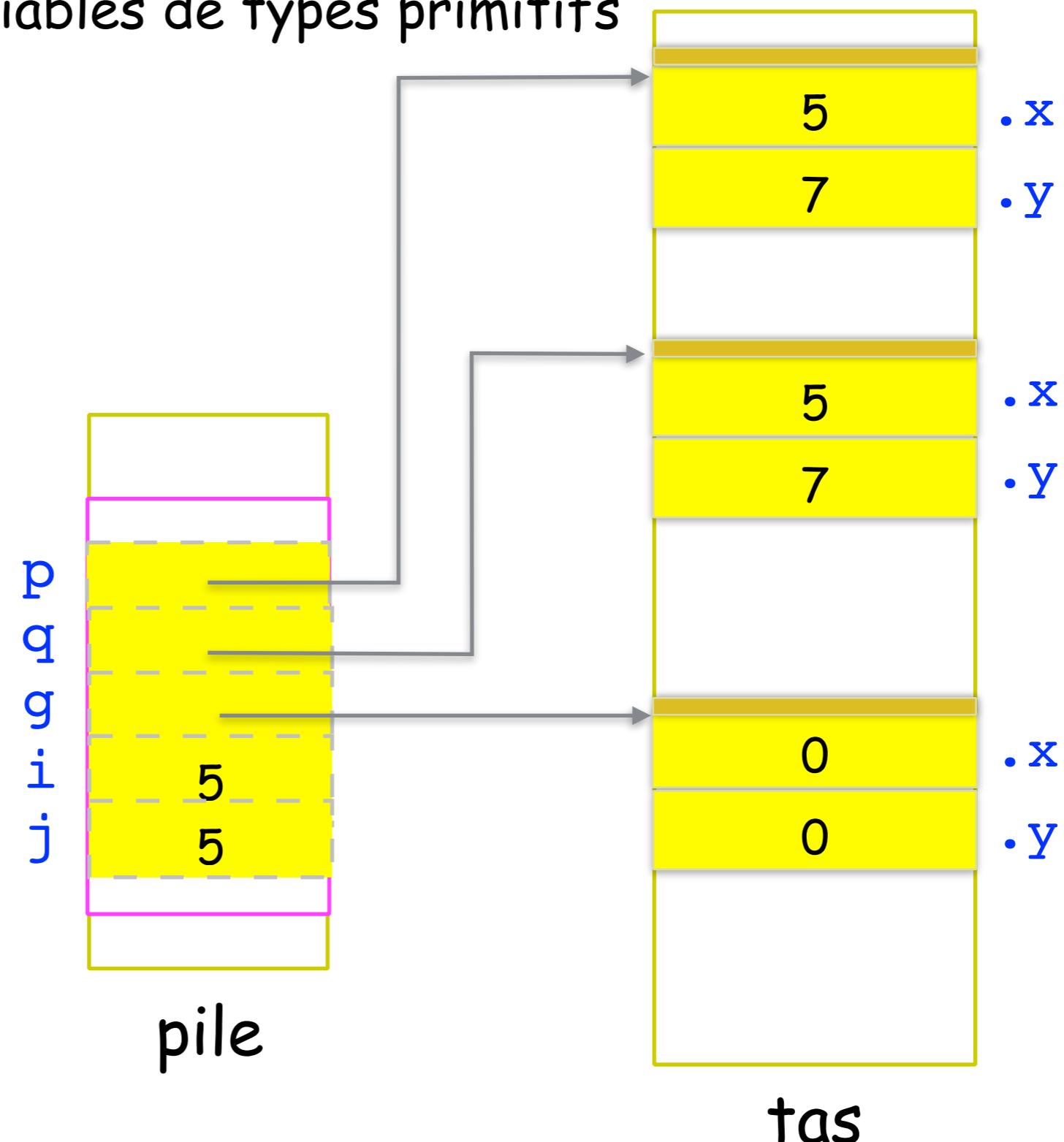


Références et objets : test d'égalité

- 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

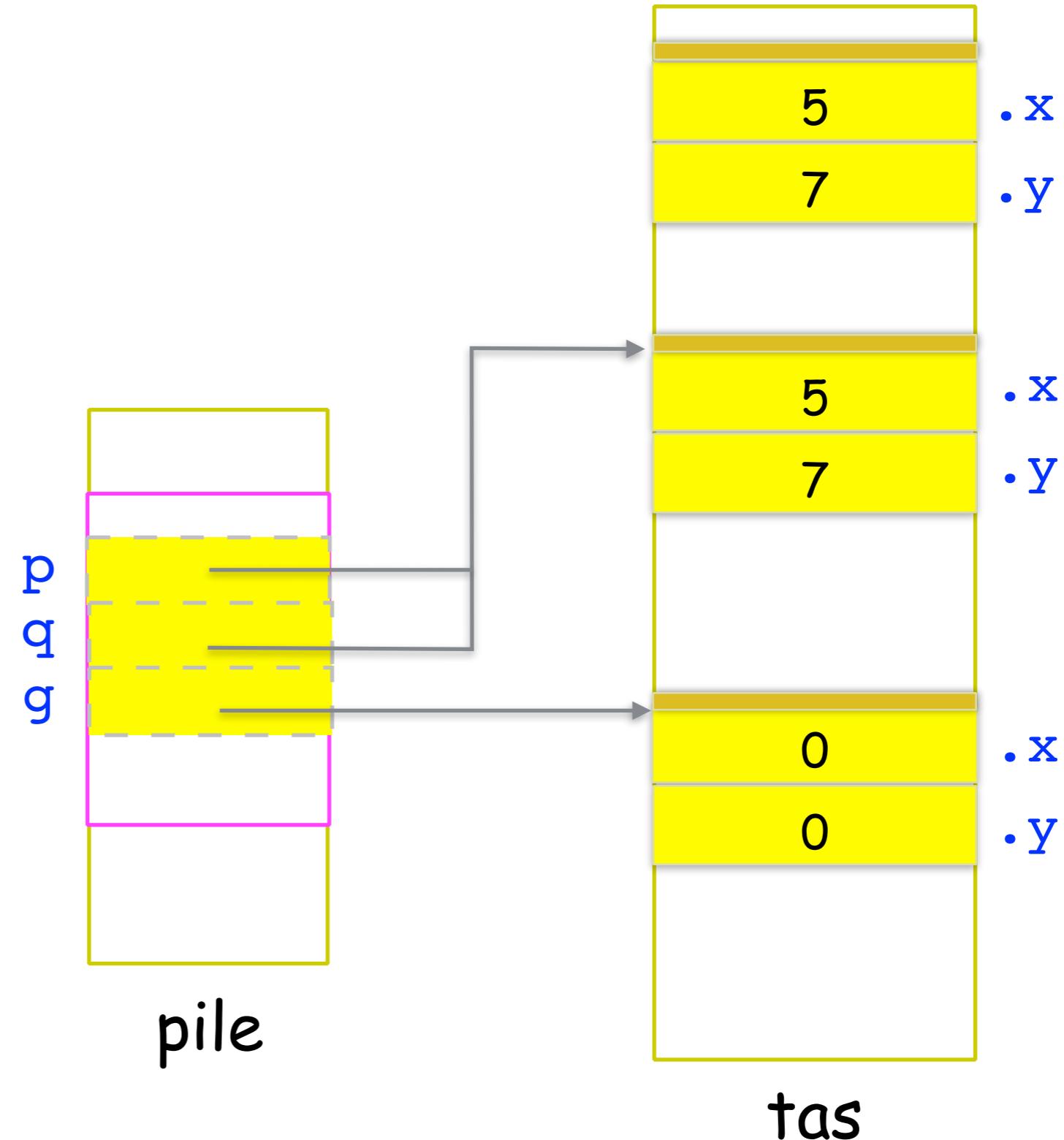


Références et objets : test et affectations

```
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

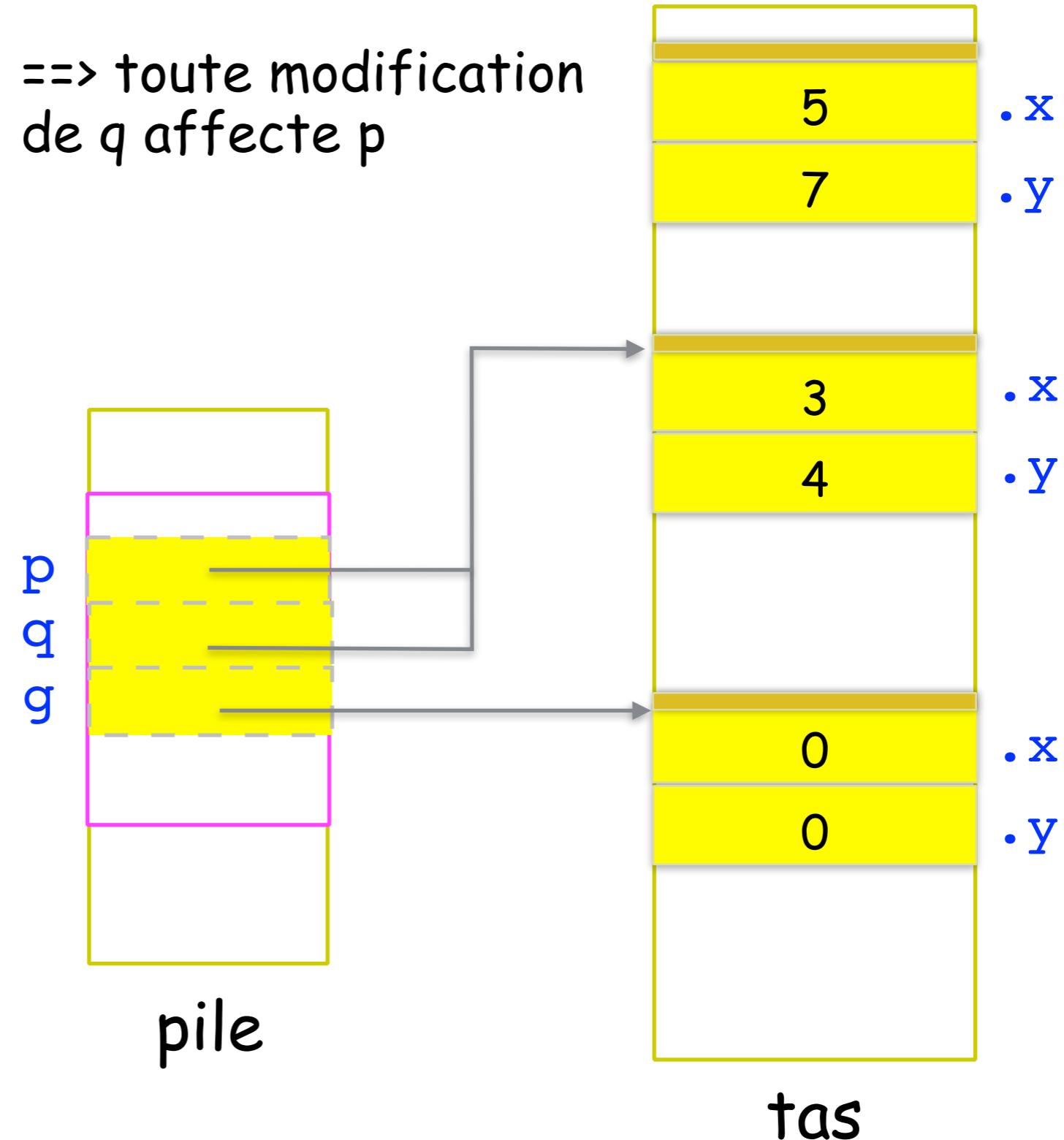


Références et objets : test et affectations

```
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

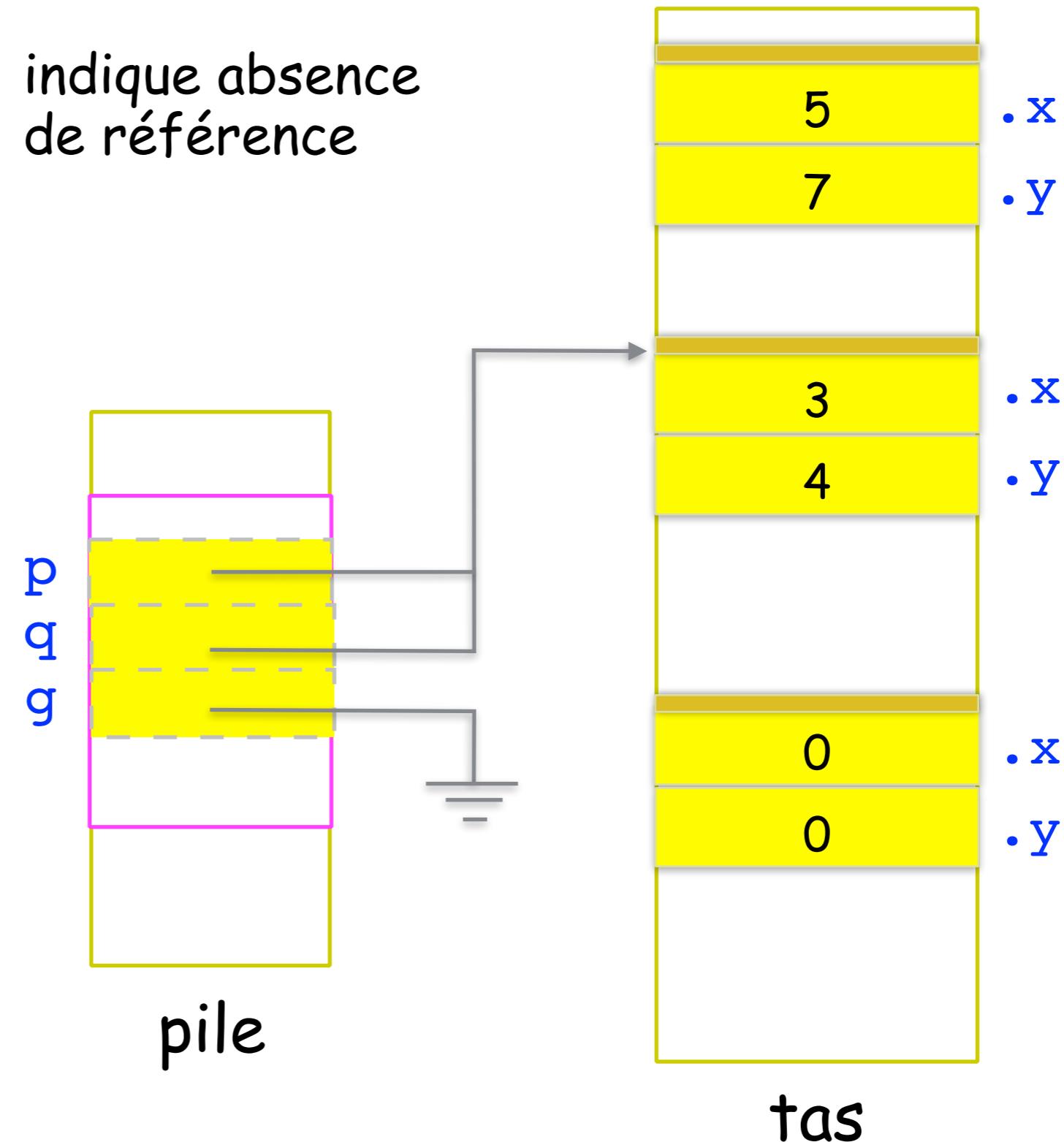


Références et objets : la constante null

```
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



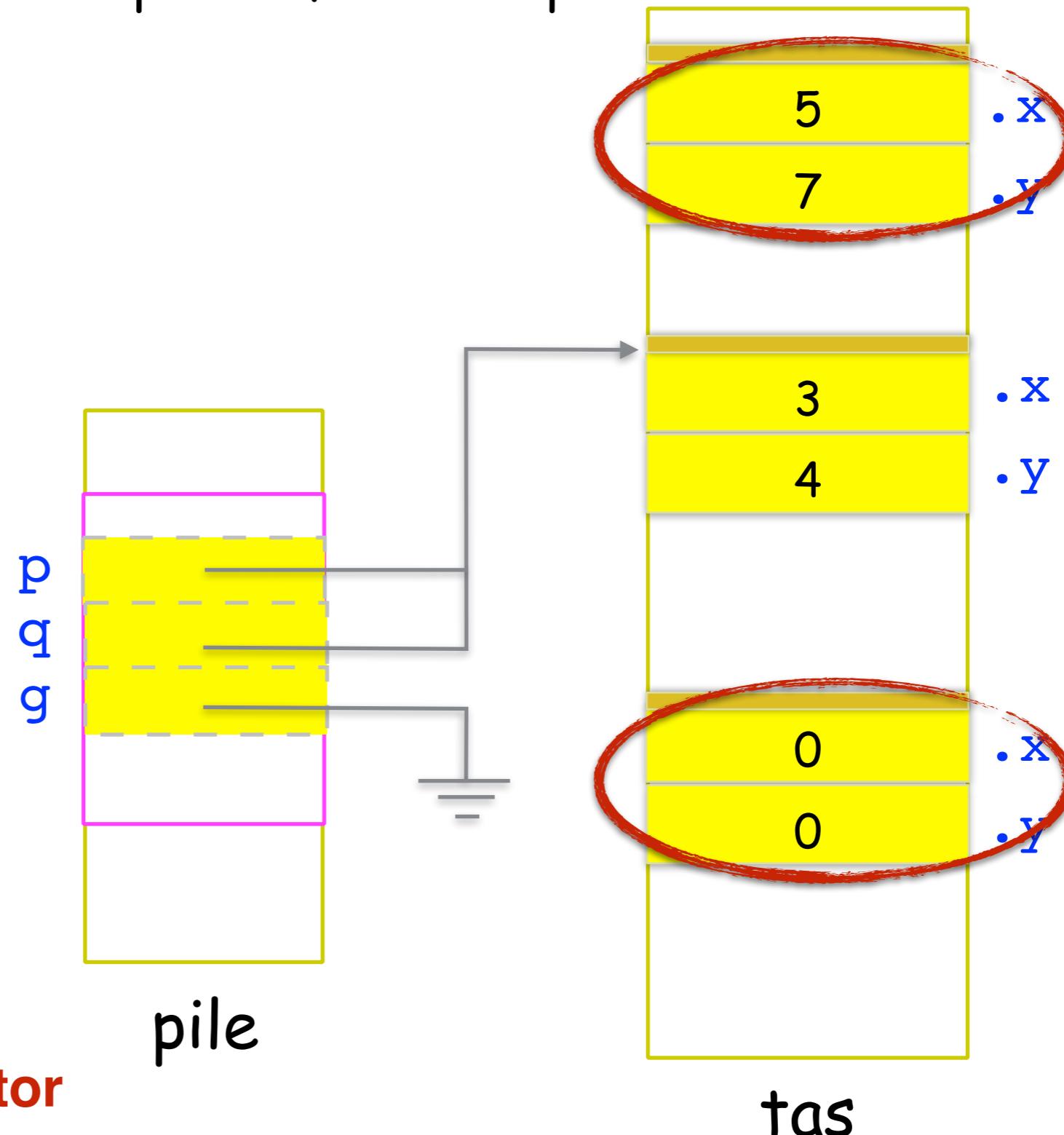
Références et objets : garbage collector

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  
    ...  
}
```

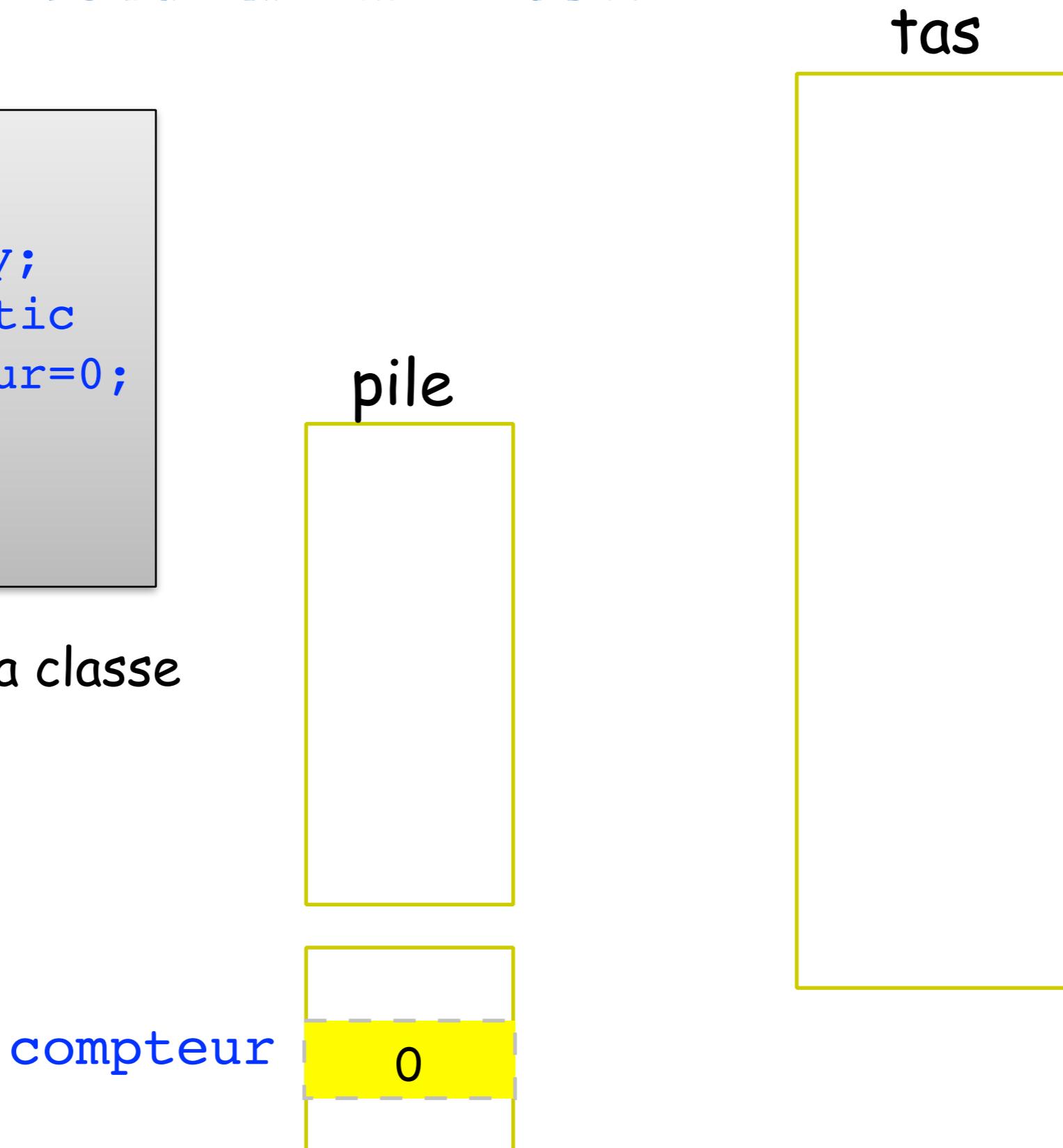
- **x, y :**
variables d'instance
 - chaque objet de la classe a sa propre copie de x et y
- **compteur :**
variable de classe
 - il y a une seule variable compteur , partagée par tous les objets de la classe

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

Champs "static" (variables de classe)

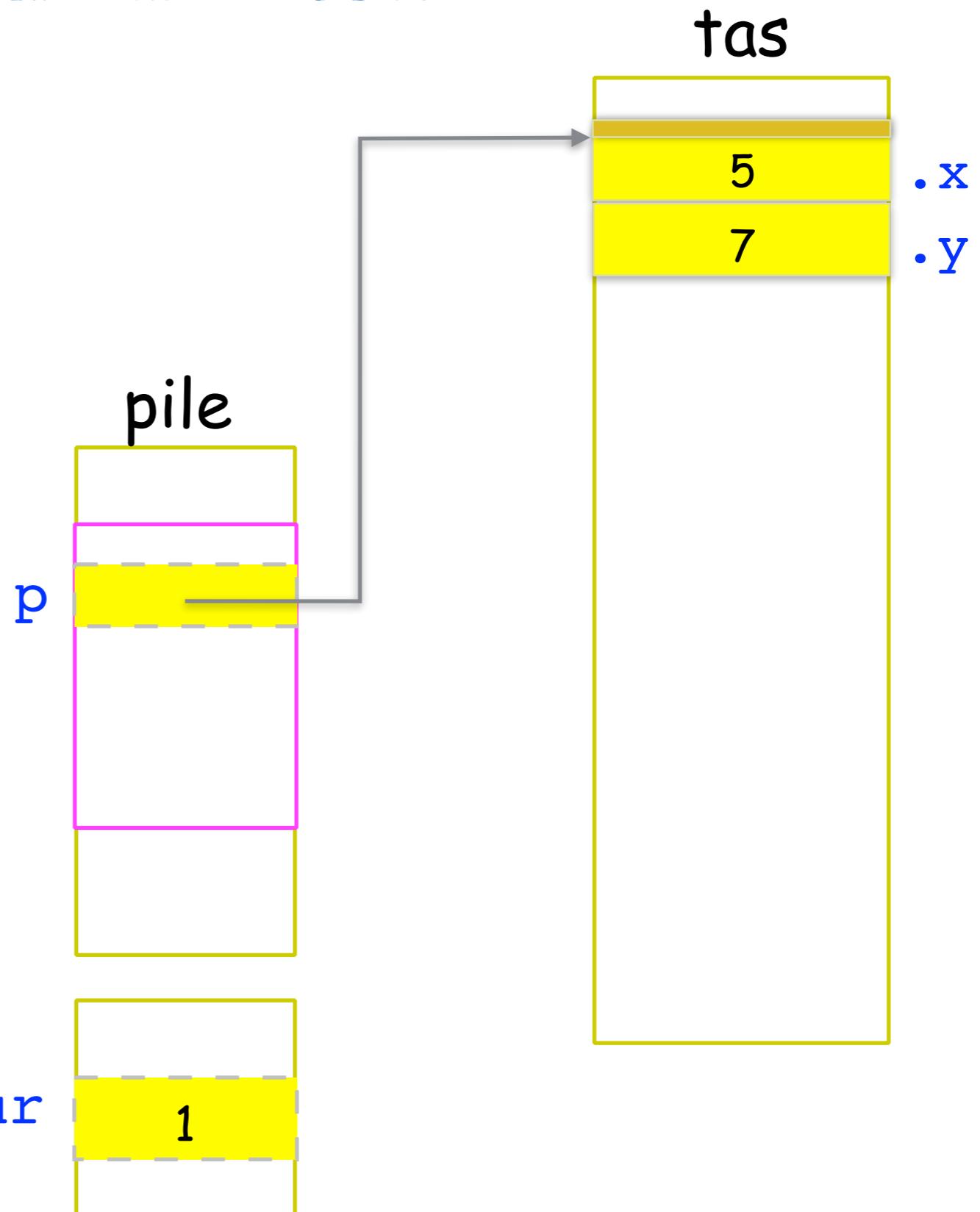
```
class Point{  
    private  
        double x, y;  
    private static  
        int compteur=0;  
    ...  
};
```

Chargement de la classe



Champs "static" (variables de classe)

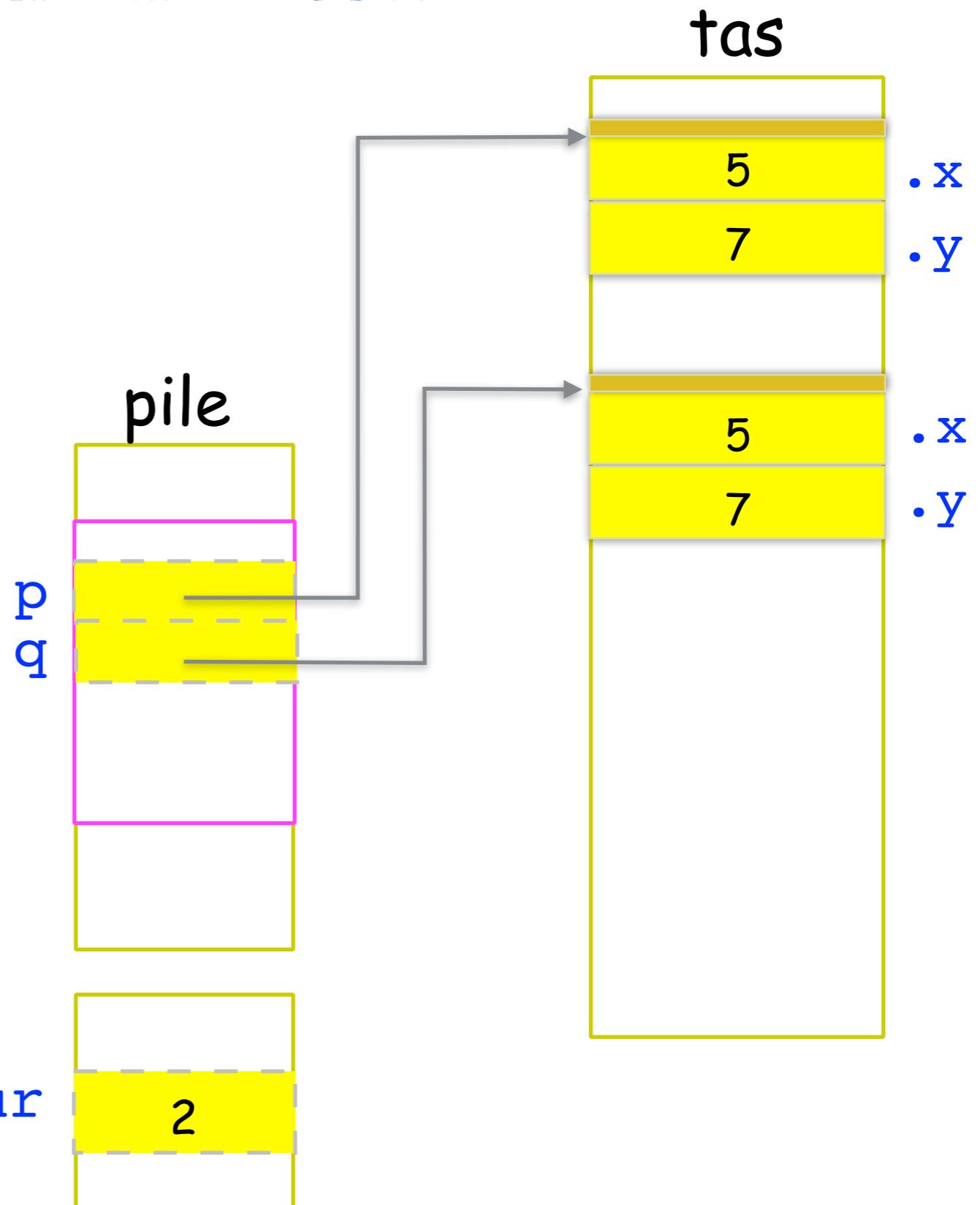
```
class Point{  
    private  
        double x, y;  
    private static  
        int compteur=0;  
    ...  
};  
...  
Point p =  
new Point (5,7);
```



Execution

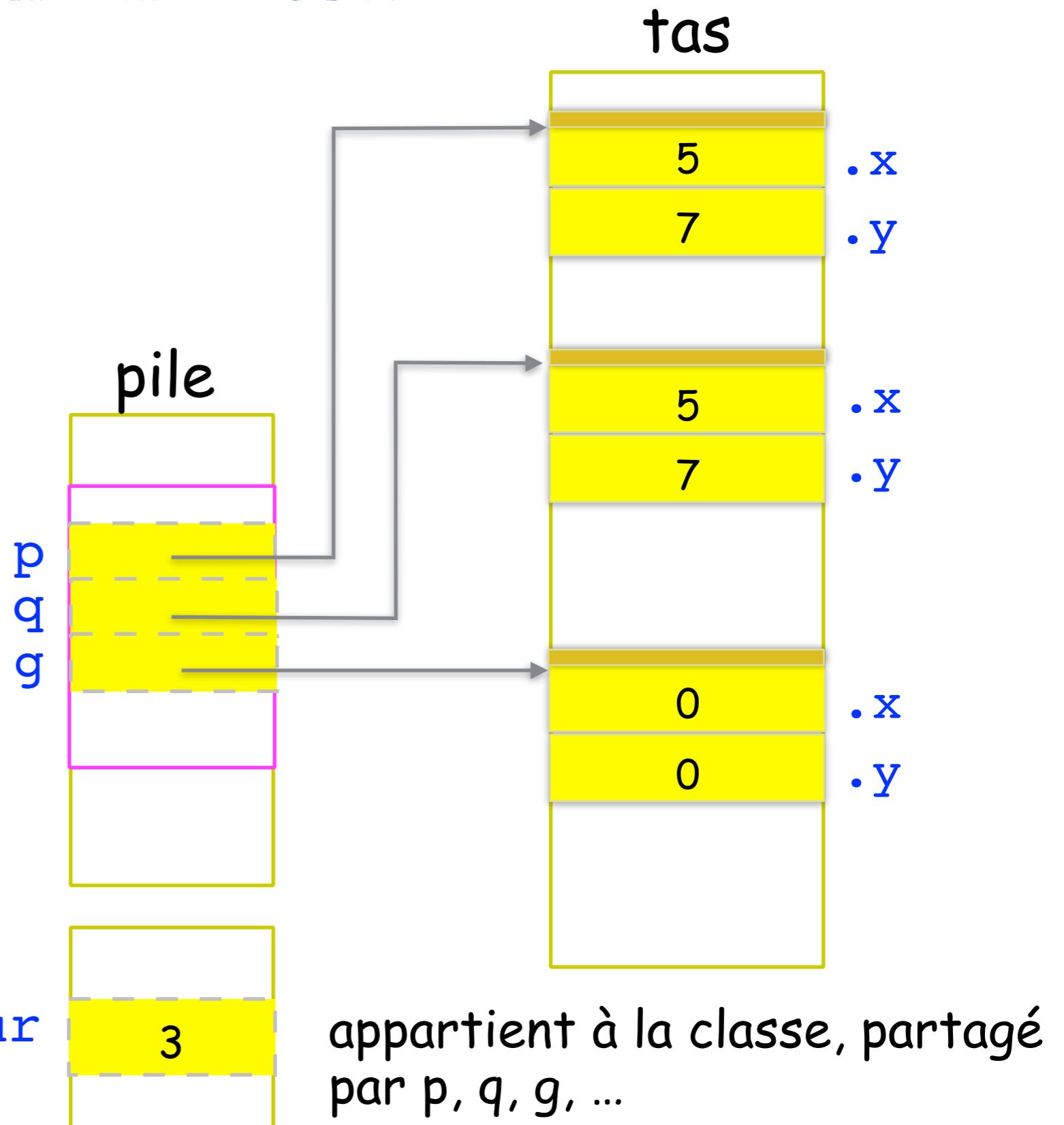
Champs "static" (variables de classe)

```
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);
```



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++; }  
    // méthodes  
    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. invocable sans qu'aucun objet soit créé)
 - 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 différents

```
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'accès**
 - **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 aussi précéder 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

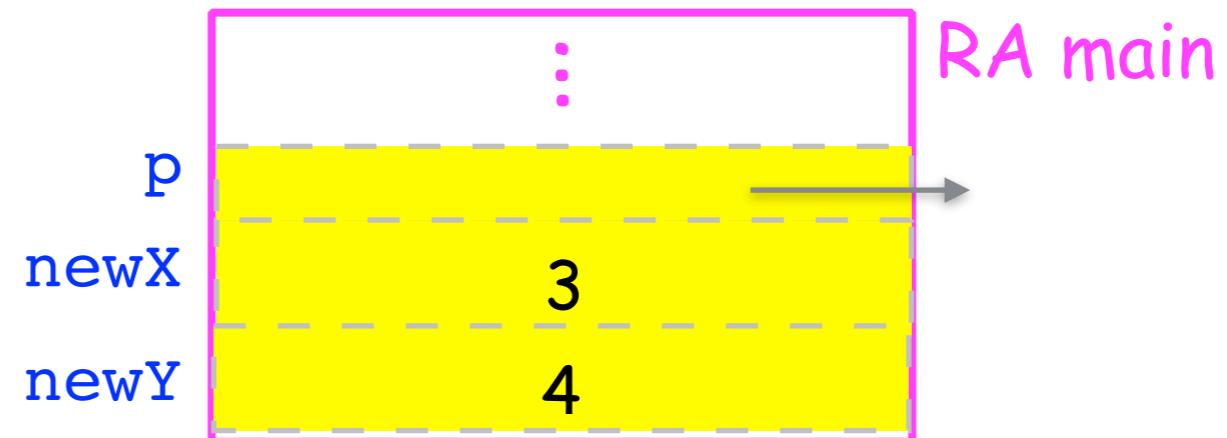
- Le passage des paramètres en Java est toujours par valeur
 - i.e. la valeur du paramètre passé est copiée dans la mémoire locale de la méthode (RA)
- Paramètres de type primitif / type référence : effet différent

```
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) { ...  
    ... }  
}
```

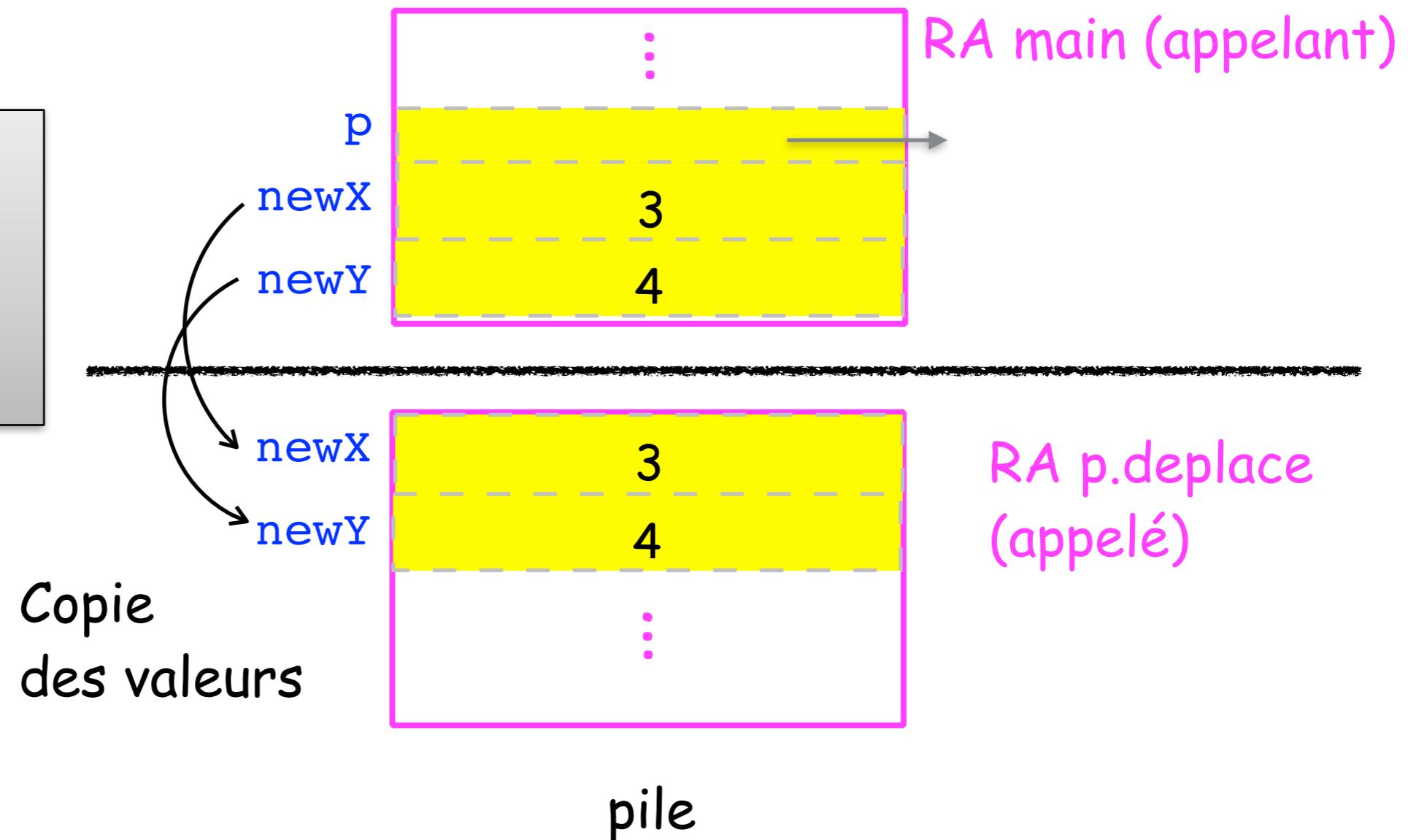
```
//main  
Point p = ...;  
newX=3; newY=4;
```



Paramètres de type primitif : exemple

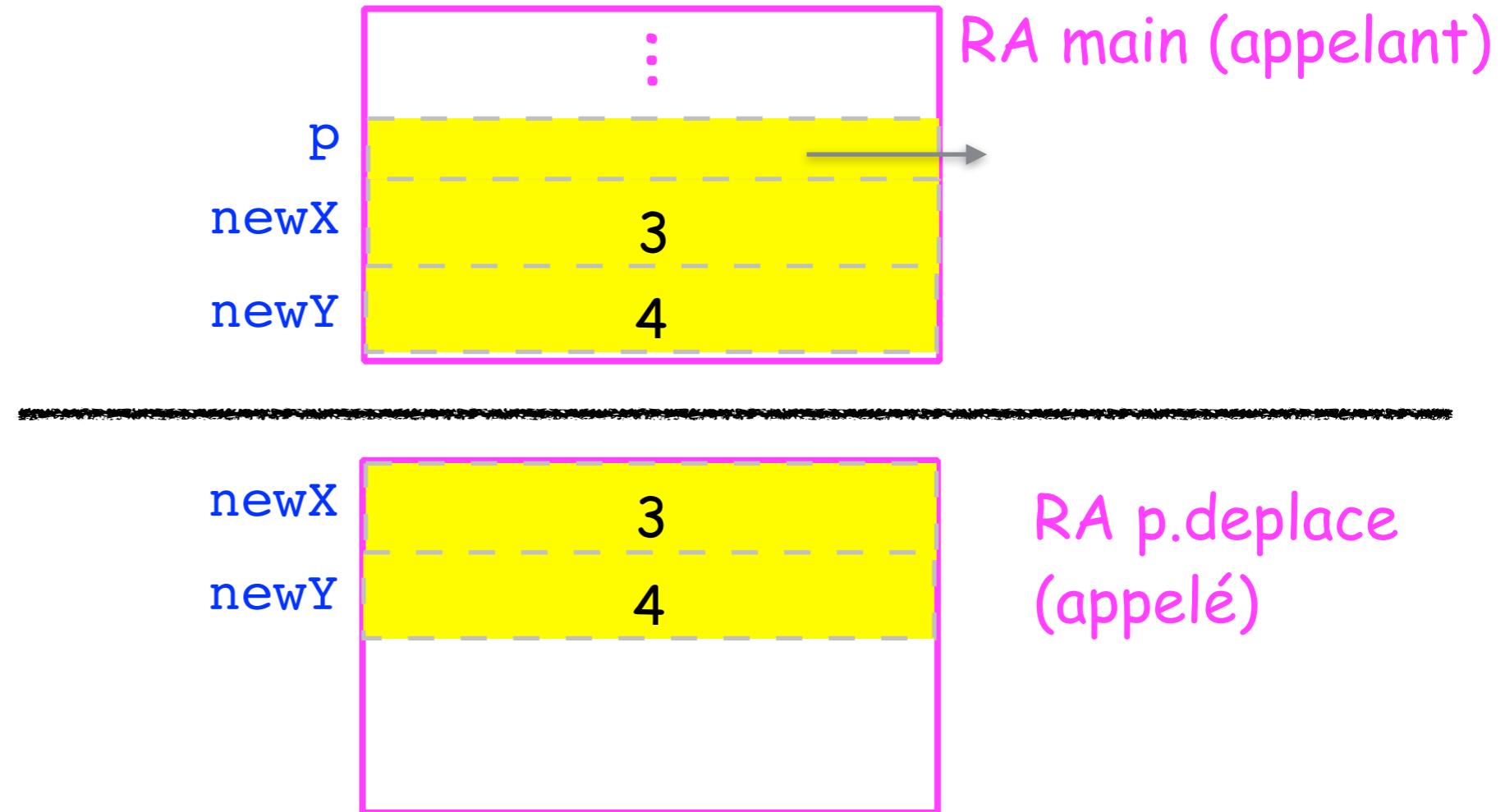
```
Class Point {  
    public void deplace (double newX, double newY) { ...  
    ... }  
}
```

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



Paramètres de type primitif : exemple

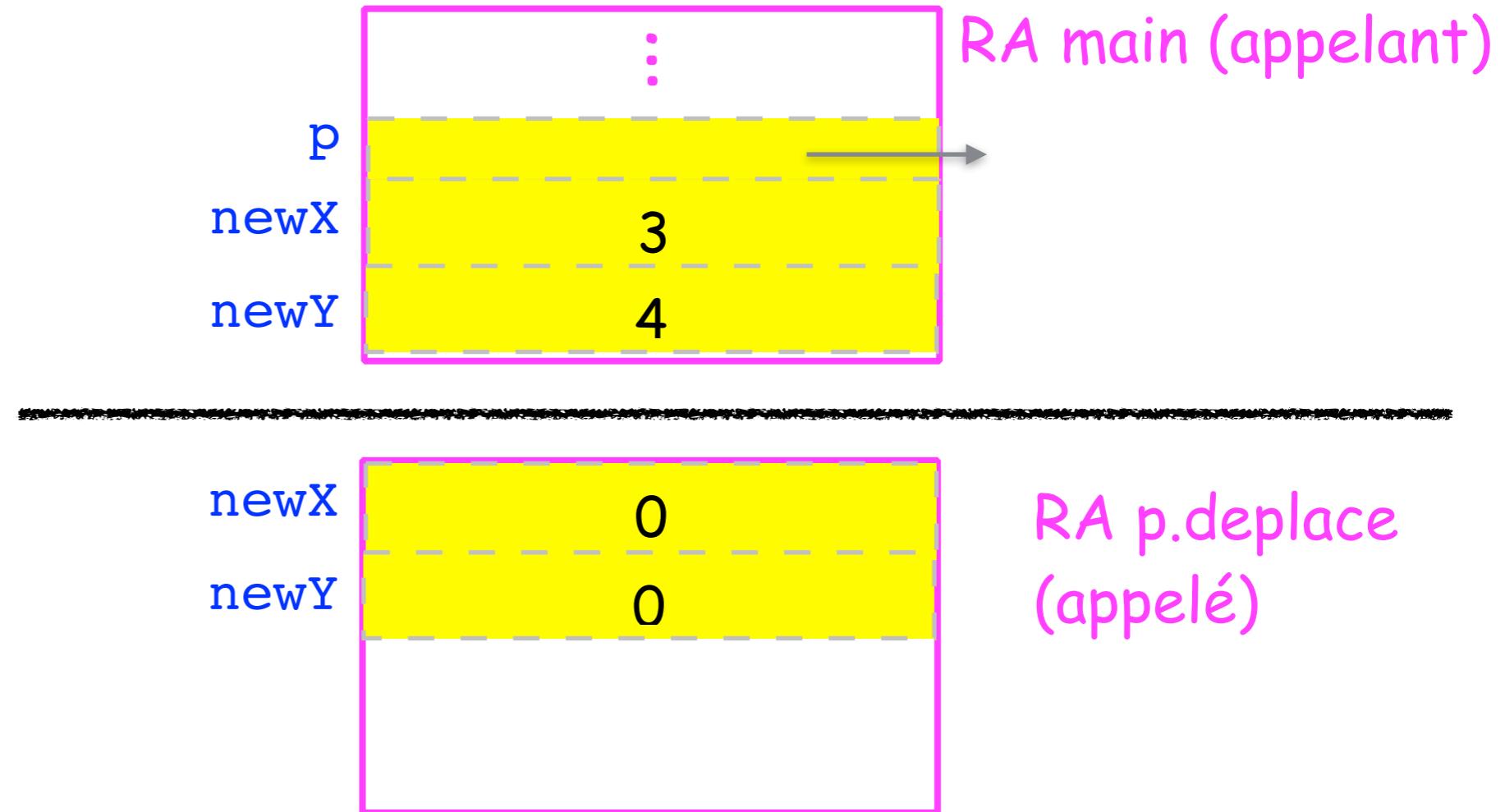
```
Class Point {  
    public void deplace (double newX, double newY) { ...  
    ... newX=0; newY=0; }  
}  
  
//main  
Point p = ...;  
newX=3; newY=4;  
p.deplace  
    (newX,newY);
```



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

Paramètres de type primitif : exemple

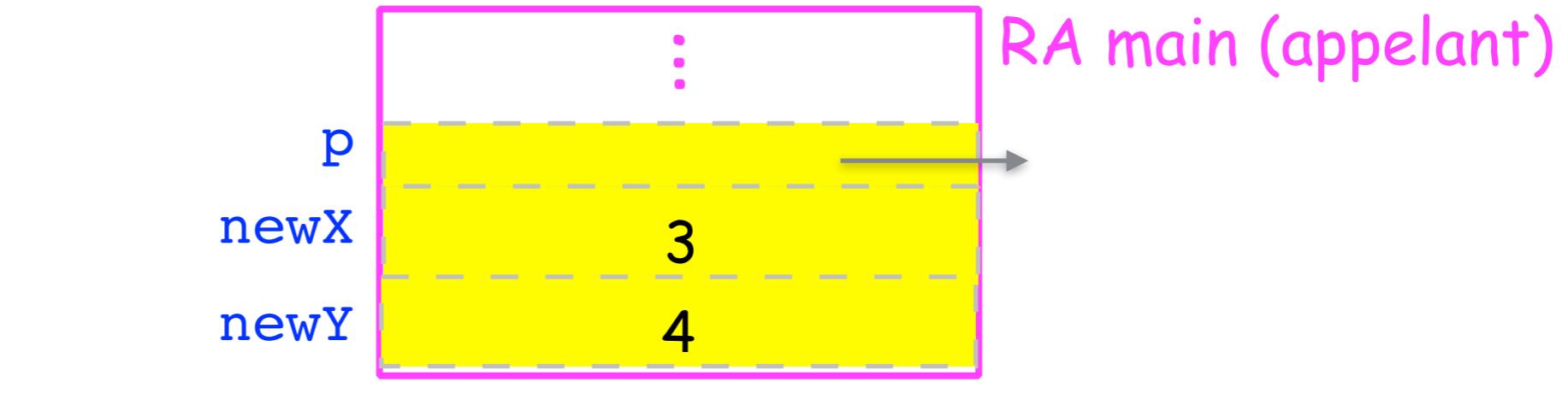
```
Class Point {  
    public void deplace (double newX, double newY) { ...  
    ... newX=0; newY=0; }  
}  
  
//main  
Point p = ...;  
newX=3; newY=4;  
p.deplace  
    (newX,newY);
```



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

Paramètres de type primitif : exemple

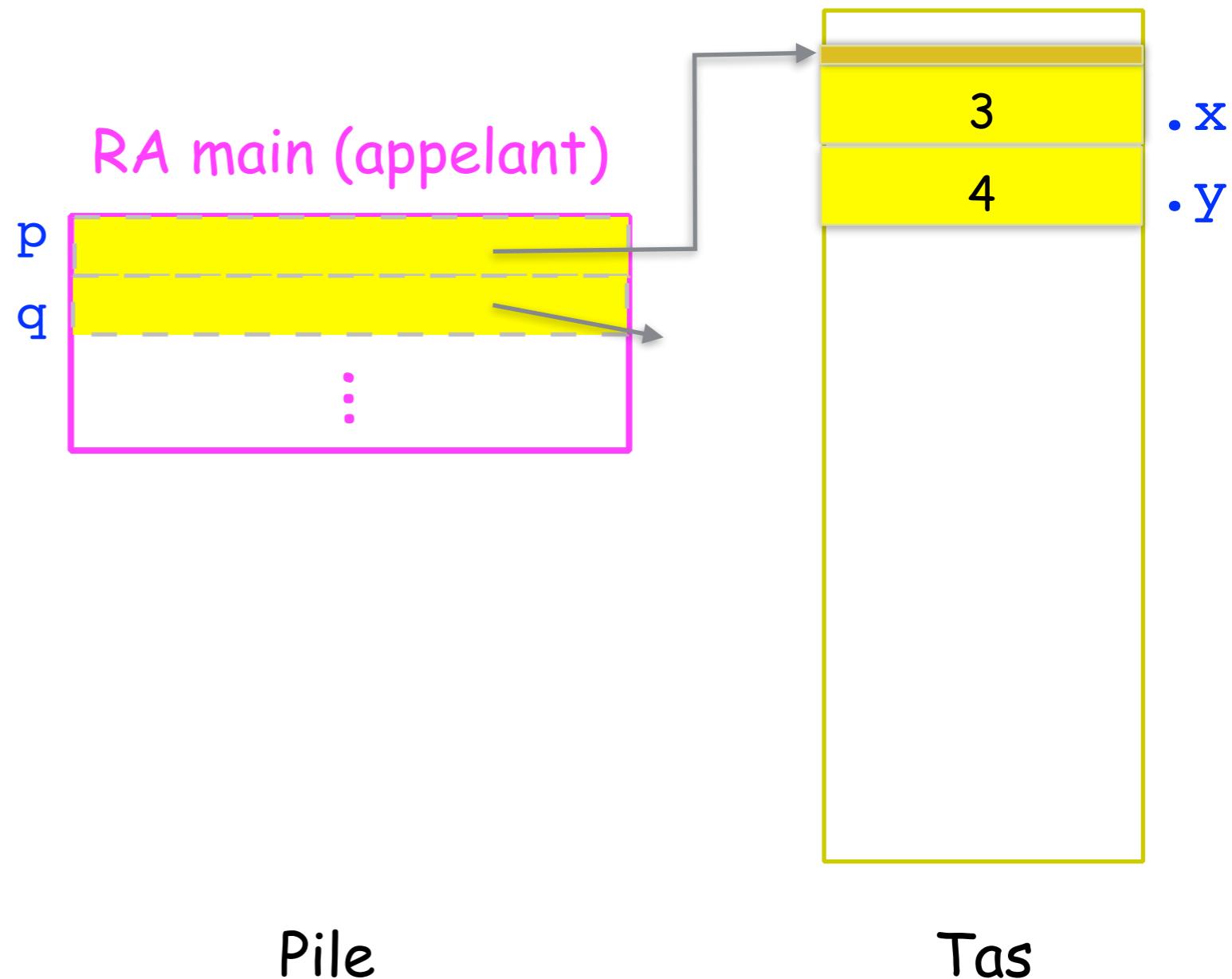
```
Class Point {  
    public void deplace (double newX, double newY) { ...  
    ... newX=0; newY=0; }  
}  
  
//main  
Point p = ...;  
newX=3; newY=4;  
p.deplace  
    (newX,newY);  
System.out.print  
(newX); //3
```



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

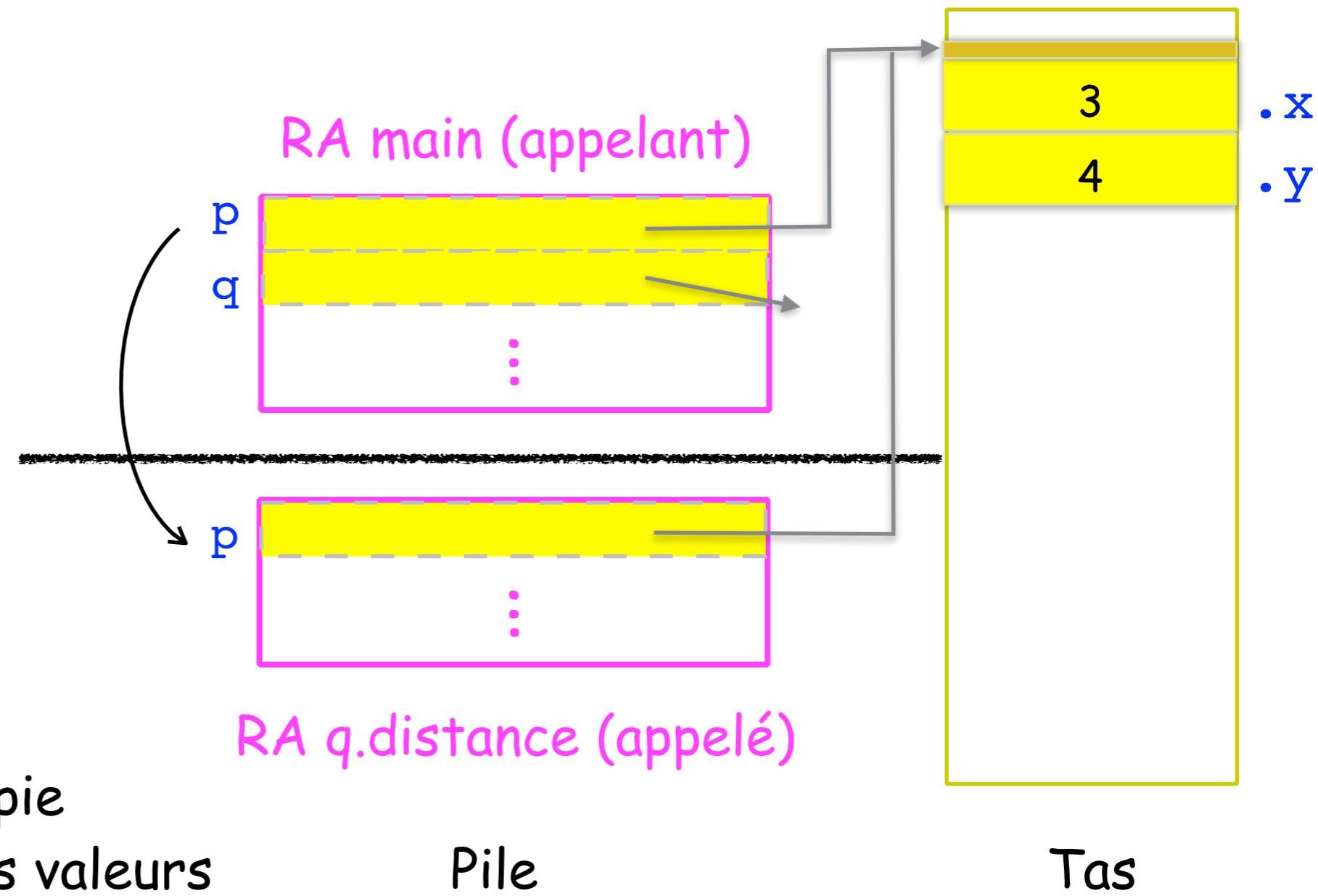
Paramètres de type référence : exemple

```
Class Point {  
    public double distance (Point p) { ...  
    ... }  
}  
  
//main  
Point p = new  
    Point(3,4);  
Point q =...;
```



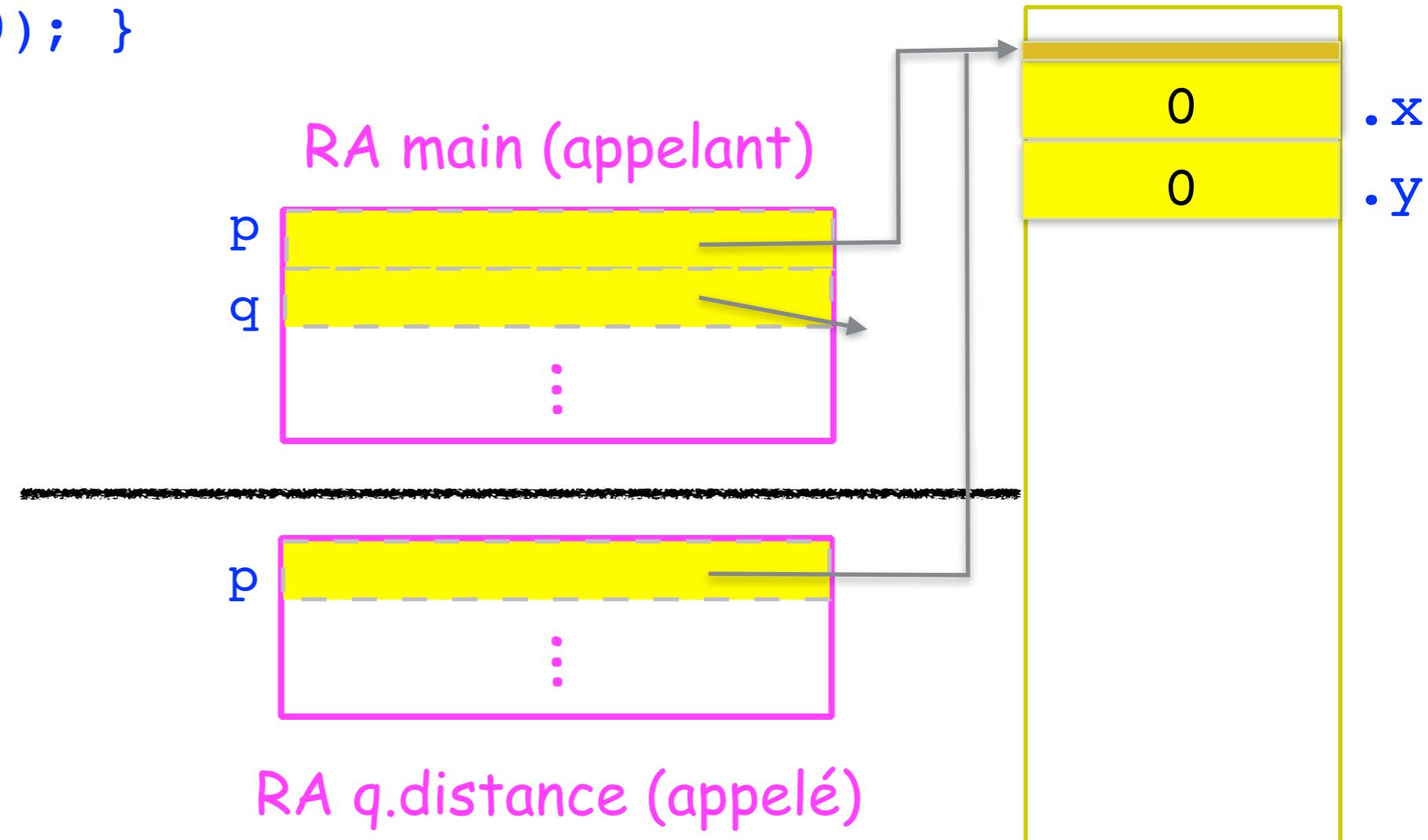
Paramètres de type référence : exemple

```
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

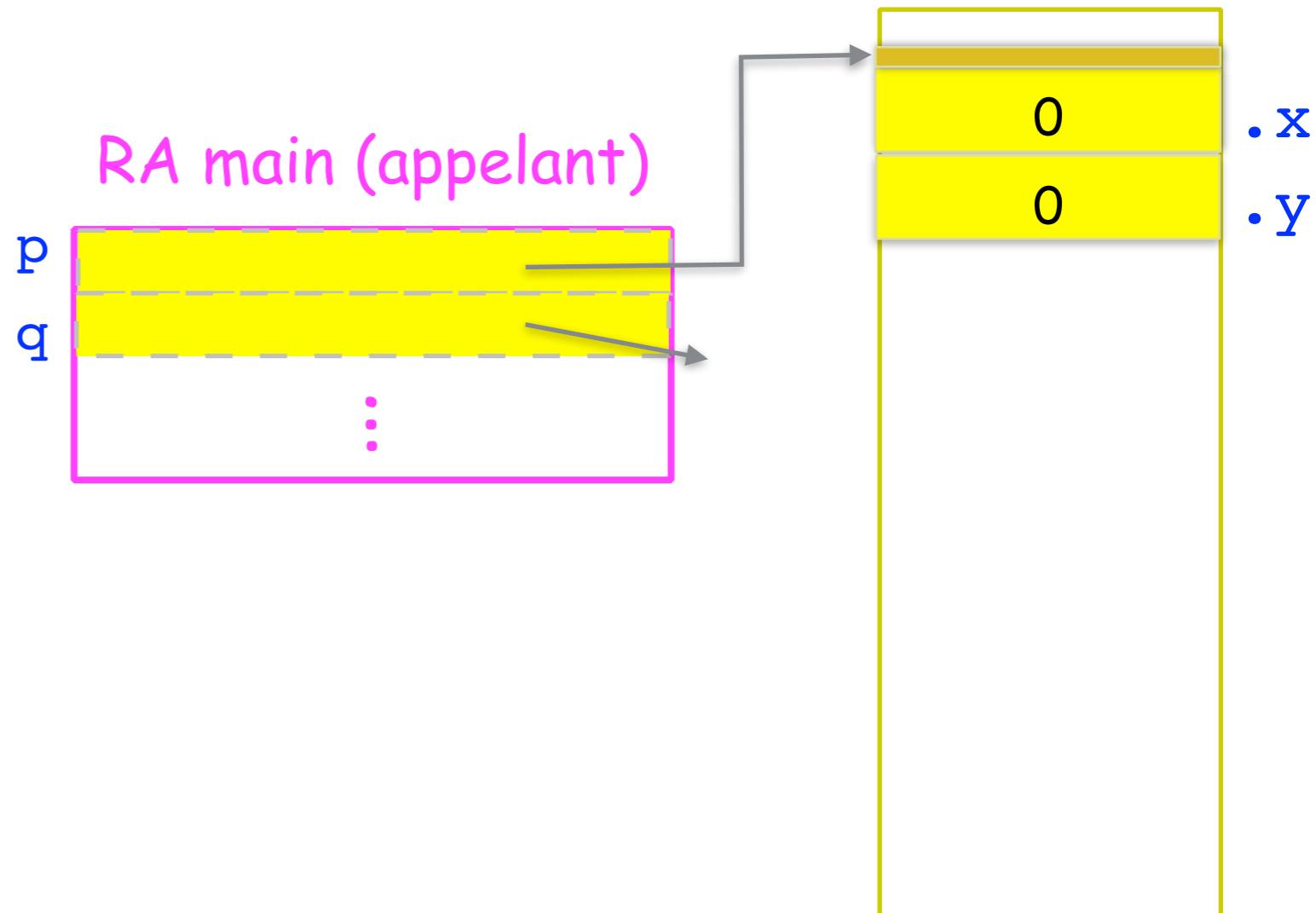
```
Class Point {  
    public double distance (Point p) { ...  
    ... p.deplace(0,0); }  
}  
  
//main  
Point p = new  
    Point(3,4);  
Point q =...;  
System.out.print  
(q.distance(p));
```



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

Paramètres de type référence : exemple

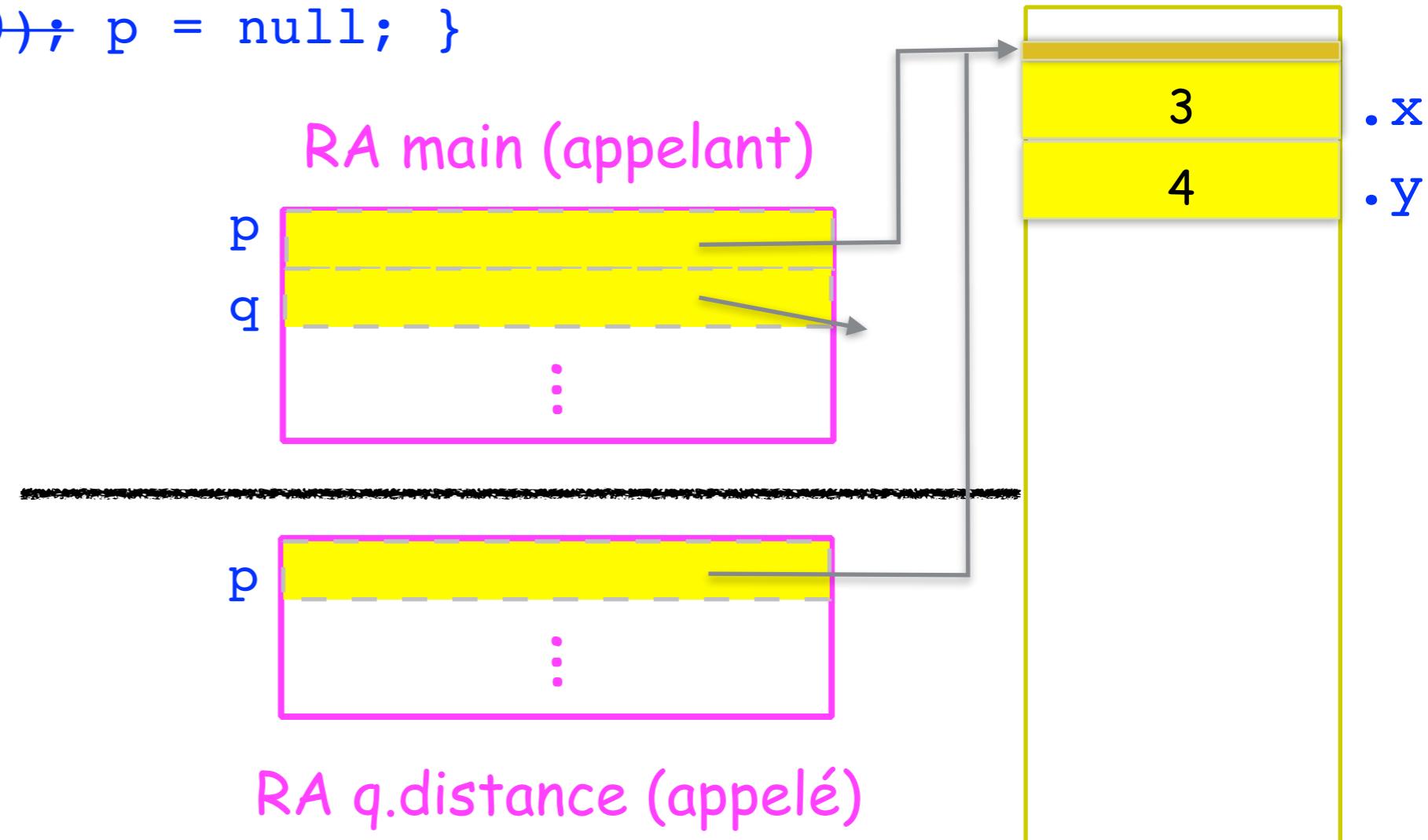
```
Class Point {  
    public double distance (Point p) { ...  
    ... p.deplace(0,0); }  
}  
  
//main  
Point p = new  
    Point(3,4);  
Point q =...;  
System.out.print  
(q.distance(p));  
System.out.print  
(p.getX()); //0
```



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

Paramètres de type référence : exemple

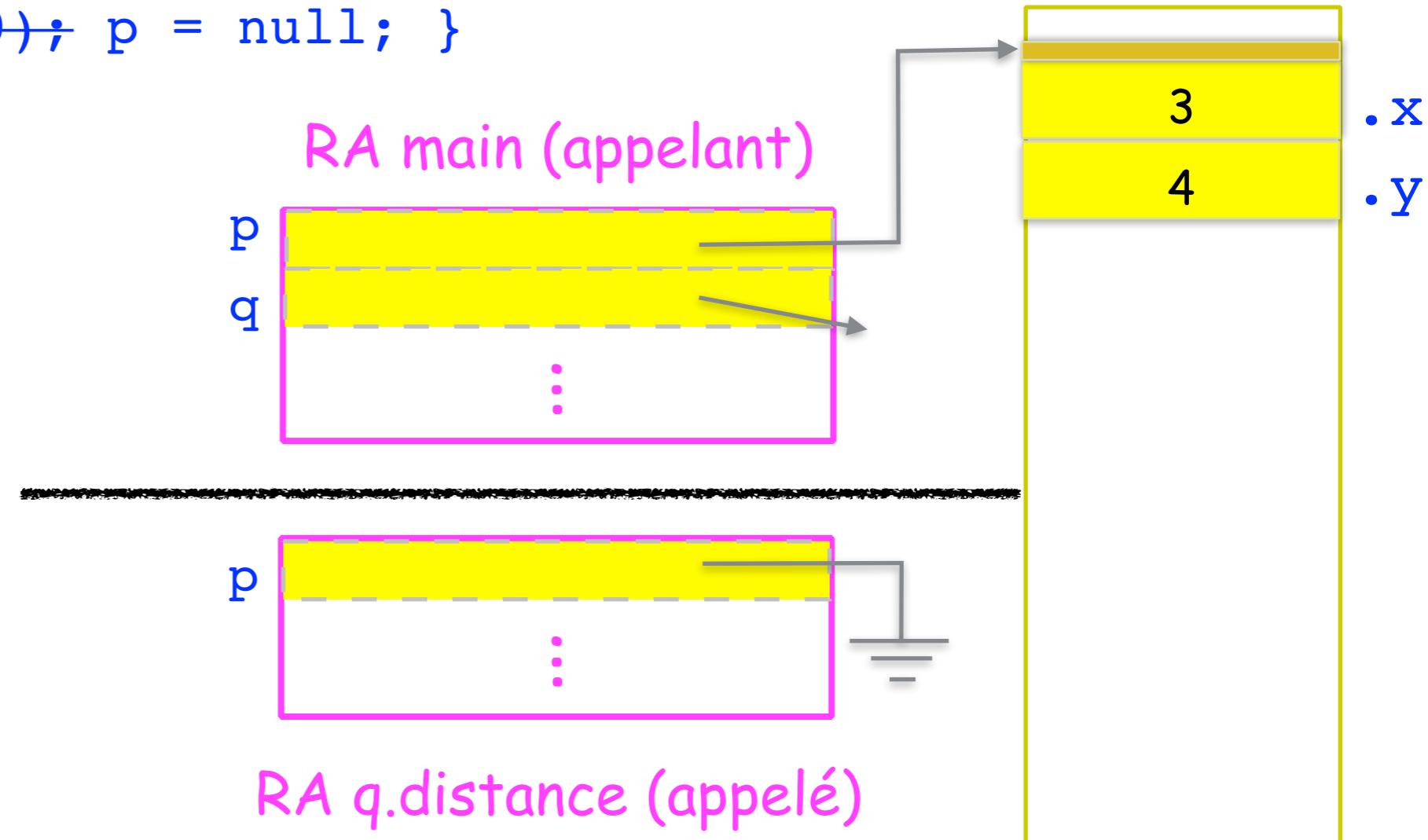
```
Class Point {  
    public double distance (Point p) { ...  
    ... p.deplace(0,0); p = null; }  
}  
  
//main  
Point p = new  
    Point(3,4);  
Point q =...;  
System.out.print  
(q.distance(p));
```



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

Paramètres de type référence : exemple

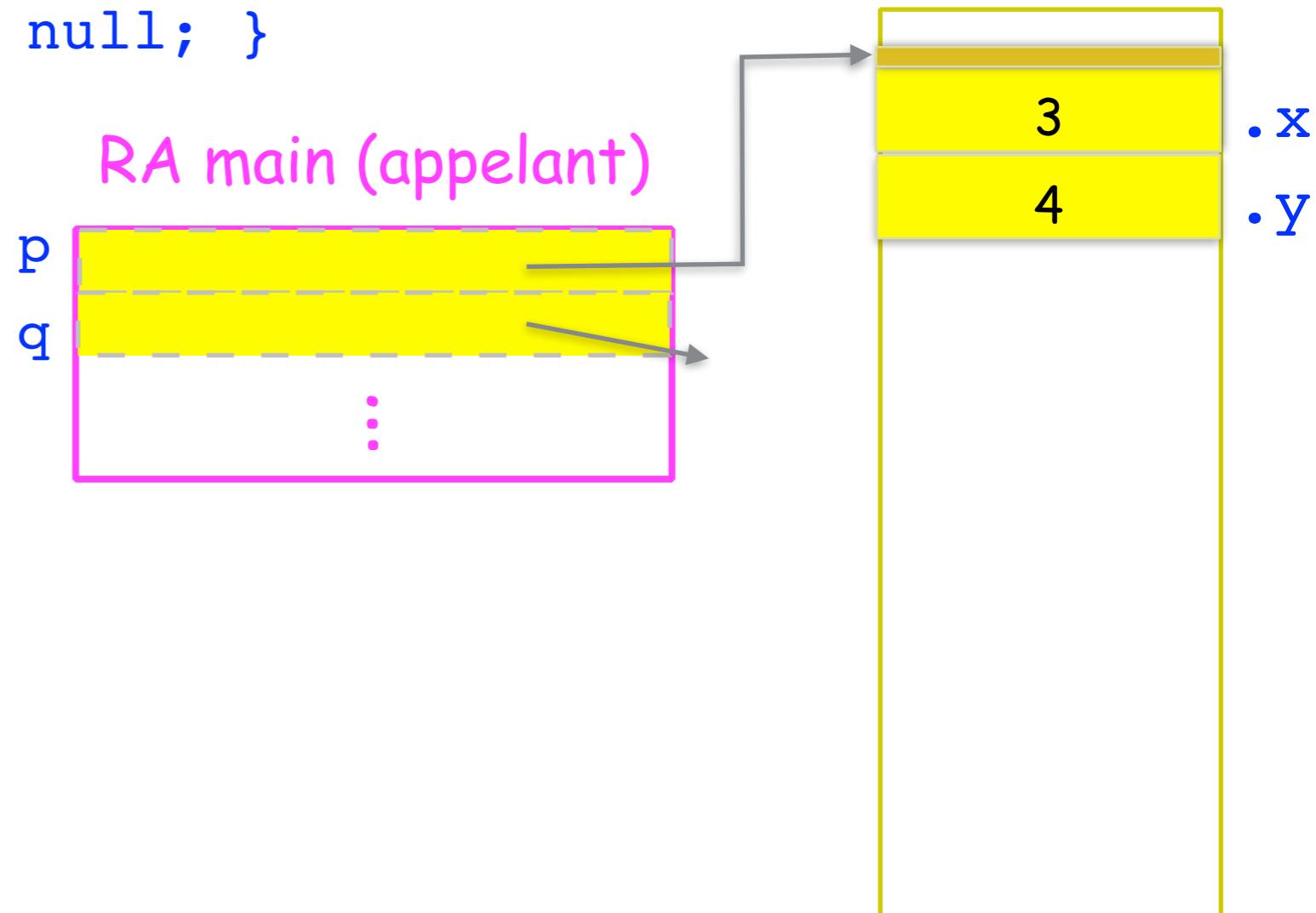
```
Class Point {  
    public double distance (Point p) { ...  
    ... p.deplace(0,0); p = null; }  
}  
  
//main  
Point p = new  
    Point(3,4);  
Point q =...;  
System.out.print  
(q.distance(p));
```



- 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

Initialisations

- Alternatives pour l'initialisation des champs :

```
class A {  
    private static long nextId = 0;  
    private int x = 1;                                initialisateur  
    private long idNum;  
    {  
        idNum = nextId++;                            bloc d'initialisation  
    }  
}
```

```
public A(int x){ this.x = x;}  
public A () {}  
//methodes  
...  
}
```

- permettent de regrouper les parties communes à tous les constructeurs

Ordre d'initialisation

- Quand un constructeur est invoqué, l'objet est déjà créé et ses champs initialisés aux valeurs par défaut
 - champs numériques : `0`
 - champs référence : `null`
 - champs booléen : `false`
- Invocation d'un constructeur* :
 1. - Si le constructeur invoque `this(...)` : Invocation (réursive) de l'autre constructeur.
- Sinon exécution des initialiseurs et des blocs d'initialisation de la classe, dans l'ordre de définition

```
class A { ...
    int i =1;
    ...
    { i++; }
}
```

2. Execution du corps du constructeur (`this()` n'en fait pas partie)

* en l'absence d'héritage

Ordre d'initialisation : exemple

```
class B {  
    private static long nextId = 0;  
    private int x = 1;  
    private int y, z, w;  
    private long idNum;  
    {idNum = nextId++;}  
    public B (int y, int z)  
    { this.y = y; this.z = z; }  
    public B (int y)  
    { this(y, 2); w = 3; }  
}  
  
//main  
B b = new B(4,5);  
  
//b.x == 1, b.y == 4  
//b.z == 5, b.w == 0  
//b.idNum = 0
```

0. initialisation par défaut
b.x, b.y, b.z, b.w : 0, b.idNum : 0
1. initialisateurs et blocs d'init.
b.x:1, b.idNum : 0, B.nextId : 1
2. corps du constructeur :
b.y : 4, b.z: 5

Ordre d'initialisation : exemple

```
class B {  
    private static long nextId = 0;  
    private int x = 1;  
    private int y, z, w;  
    private long idNum;  
    {idNum = nextId++;}  
    public B (int y, int z)  
    { this.y = y; this.z = z; }  
    public B (int y)  
    { this(y, 2); w = 3; }  
}  
  
//main  
B b = new B(4,5);  
B c = new B(6);  
  
//c.x == 1, c.y == 6  
//c.z == 2, b.w == 3  
//b.idNum = 1
```

0. initialisation par défaut
c.x, c.y, c.z, c.w : 0, c.idNum :0
1. invocation de this(6,2)
 - 1.1. initialisateurs et blocs d'init.
c.x:1, c.idNum : 1, B.nextId : 2
 - 1.2. corps du constructeur :
c.y : 6, c.z: 2
2. corps du constructeur
c.w : 3

Blocs d'initialisation static

```
public class Puissancedeux {  
    static int[] tab = new int[12];  
    static {  
        tab[0]=1;  
        for(int i = 0; i < tab.length-1; i++)  
            tab[i+1]= suivant(tab[i]);  
    }  
    static int suivant(int i){  
        return i*2;  
    }  
}
```

- Au chargement de la classe : création des champs static
 - Exécution des initialisateurs static et des blocs d'initialisation static dans l'ordre de définition

Surcharge (overloading)

- **Signature d'une méthode**: le nom de la méthode et les types des paramètres (mais pas le type de la valeur renvoyée)
- Dans une classe : **pas deux méthodes avec la même signature**
- **Surcharge (overloading)** : dans une même classe signatures différentes pour un même nom de fonction

```
Class A {  
    int f(int i) {...}  
    int f(double i) {...}  
    double f() {...}  
    int f (int i, int j) {...}  
} //surcharge  
Class B {  
    int f(int i) {...}  
    char f(int j) {...}  
} // interdit : les deux méthodes ont la même signature
```

- (Java, C++, Ada permettent la surcharge)

Surcharge (overloading)

- Exemple de surcharge : les constructeurs
- Autre exemple : deux méthodes distance

```
Class Point {  
    ...  
    //distance d'un point p  
    public double distance (Point p) {  
        return Math.sqrt((p.x-x)*(p.x-x)+(p.y-y)*(p.y-y));  
    }  
    //distance de l'origine  
    public double distance() {  
        return Math.sqrt(x*x + y*y);  
    }  
}
```

- En phase de compilation, la méthode à appeler est déterminée en analysant le type des paramètres passés (**resolution de la surcharge**)
- S'il y a ambiguïté (plusieurs fonctions candidates) : erreur de compilation

Resolution de la surcharge : ambiguïté

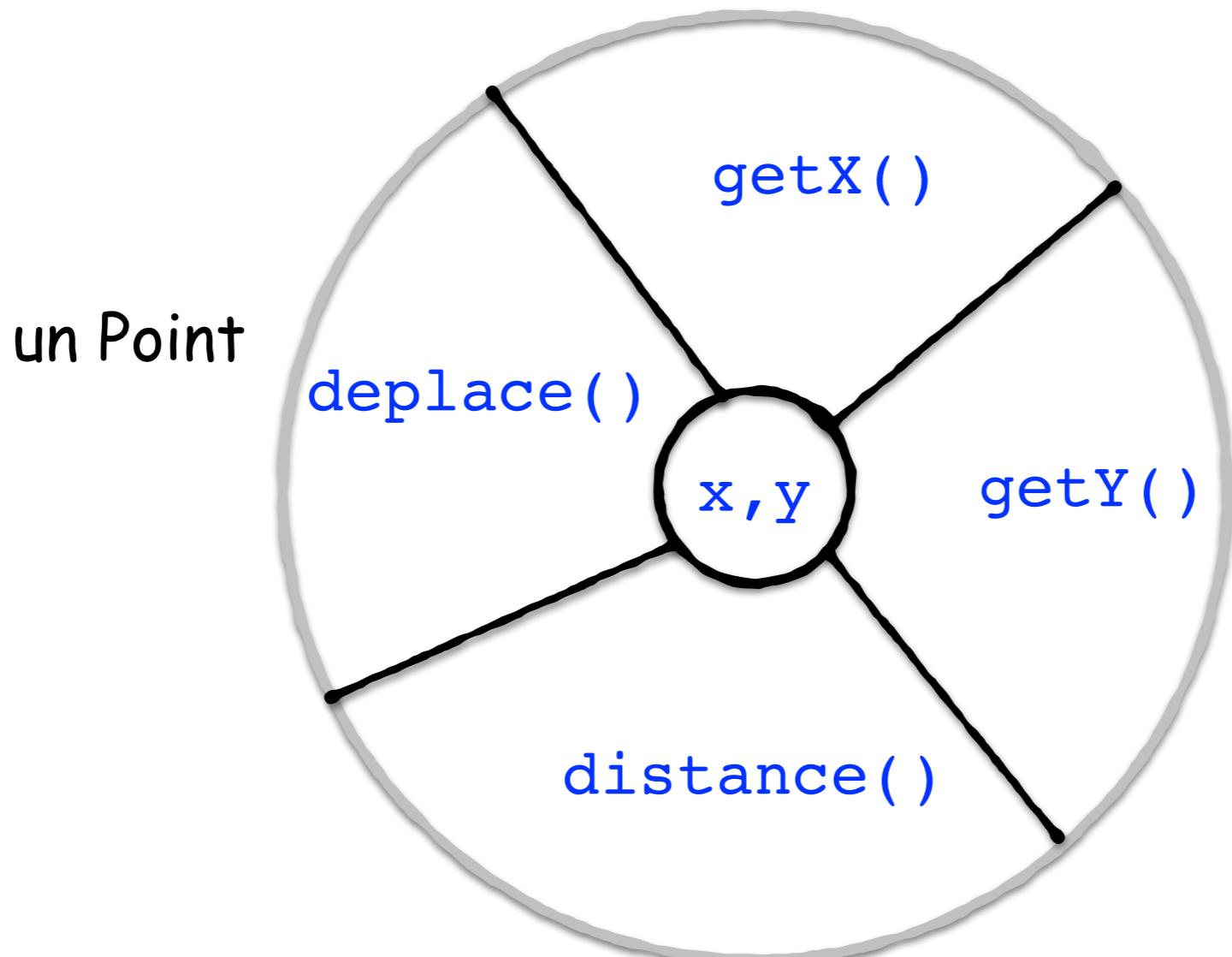
```
class Ambiguity {  
  
    void g ( int i, double d) {...  
    }  
    void g (double d, int i) {...  
    }  
    public static void main (String [ ] args) {  
        Ambiguity o = new Ambiguity();  
        int i = 5;  
        o.g(i,i); //ERREUR à la compilation  
        //deux signatures sont compatibles avec cet appel  
    }  
}
```

Resolution de la surcharge : ambiguïté

```
class WOAmbiguity {  
    void f (float i) {...}  
    void f (double i) {...}  
    public static void main (String [ ] args) {  
        Ambiguity o = new Ambiguity();  
        int i = 5;  
        o.f(i);  
        //PAS d'erreur : deux signatures sont compatibles  
        //avec cet appel, mais une est plus "proche" que l'autre  
    }  
}
```

Classes et encapsulation

- Le contrôle d'accès permet de réaliser le principe d'encapsulation
- Dans une classe typiquement :
 - champs **private** : l'implémentation des données cachée
 - méthodes **public** : interface avec laquelle manipuler les données cachées



Classes et encapsulation

- méthodes **private** :
 - utiliser pour des fonctions auxiliaires liées à l'implémentation
- méthodes de niveaux d'accès intermédiaires (**package**, **protected**)
 - pour restreindre l'accès à certaines fonctionnalités
- champs **public** :
 - à éviter, sauf dans des cas très exceptionnels
 - e.g. champs constants (**final**)
- champs de niveaux d'accès intermédiaires (**package**, **protected**)
 - utilité dans des cas spécifiques (voir plus loin)
 - à utiliser avec précaution

Classes et encapsulation

- Avantage principal de l'encapsulation : **modularité**
 - on peut utiliser une classe sans connaître les détails de son implementation
 - on peut changer l'implementation d'une classe sans changer le code qui l'utilise
 - les données restent cohérentes par rapport à la specification de la classe

Modularité : projet d'une classe Triangle

- Ayant implémenté toutes les opérations sur les Points, on peut concevoir d'autres classes, qui utilisent la classe Point
- Classe Triangle :
 - données : 3 points (pas alignés)
 - méthodes : tailles des cotés, périmètre, etc...
- La classe Triangle peut utiliser les points comme "black box"
- Structure des fichiers :
 - un fichier .java peut contenir plusieurs classes, mais une seule classe publique
 - typiquement 1 fichier par classe
 - Chaque classe qui n'est pas dans un package est visible au moins par toutes les classes dans le même répertoire (package par default)

(Plus de détails sur les packages et la visibilité plus loin)

-> `code/poo/geometry/Point.java`

`Triangle.java`

Préserver l'encapsulation

- l'état interne d'un objet doit être private
 - il est donc souvent nécessaire de fournir des méthodes public pour lire (une partie de) de l'état interne
-
- Appelés **méthodes d'accès** ("accessor methods" ou "getters")
 - Convention : les nommer **getNomChamp**

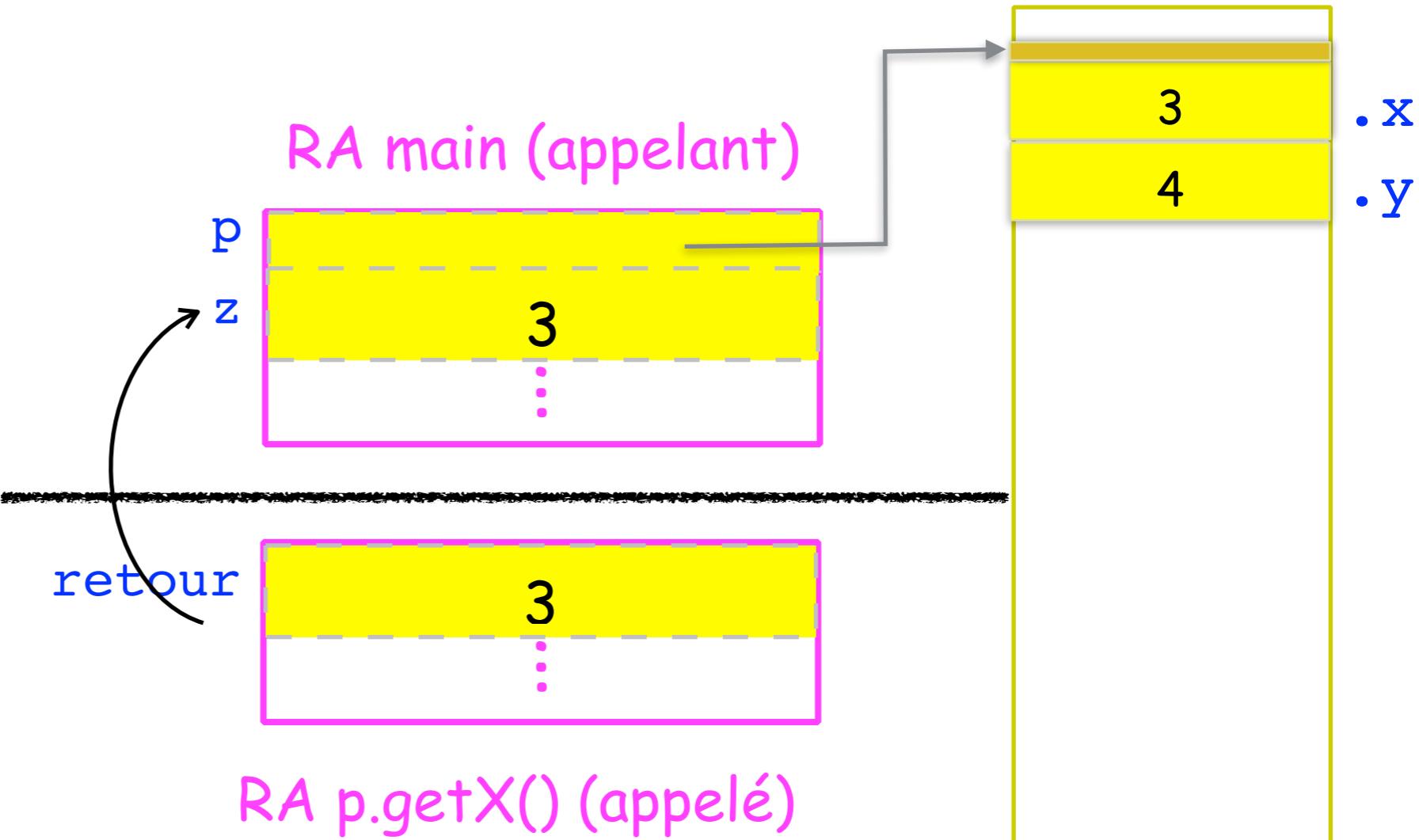
```
Class Point {  
    private double x, y;  
    public getX() { return x; }  
    public getY() { return y; }  
    ...  
}
```

OK parce que la valeur de retour est passée par valeur à l'appelant
toujours OK quand le type de retour est primitif

Préserver l'encapsulation

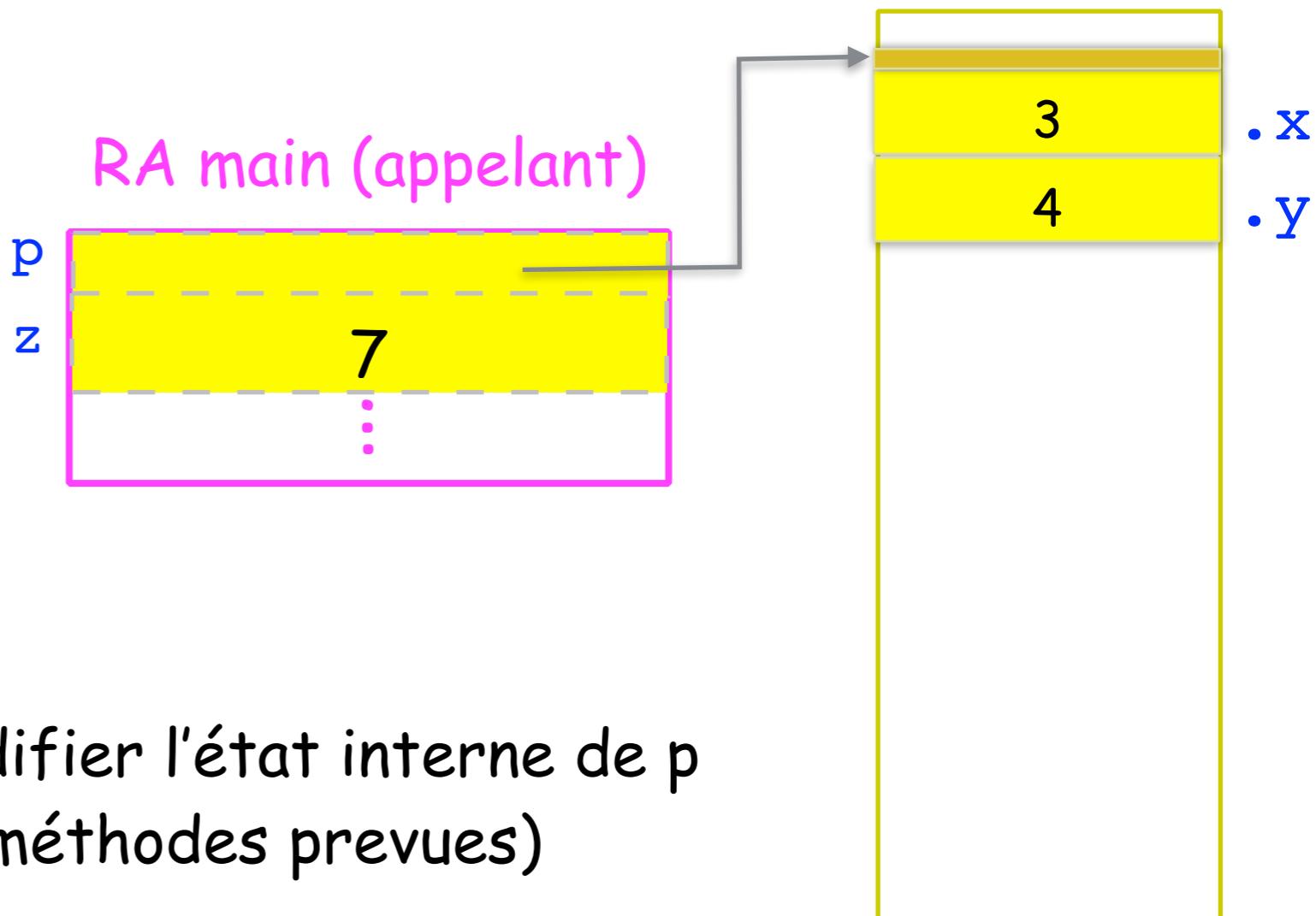
```
//main  
Point p = new  
    Point(3,4);  
int z =  
    p.getX();
```

retour par copie



Préserver l'encapsulation

```
//main  
Point p = new  
    Point(3,4);  
int z =  
    p.getX();  
z = 7;
```



OK l'appelant ne peut pas modifier l'état interne de p
(de façon autre que avec les méthodes prevues)

Préserver l'encapsulation

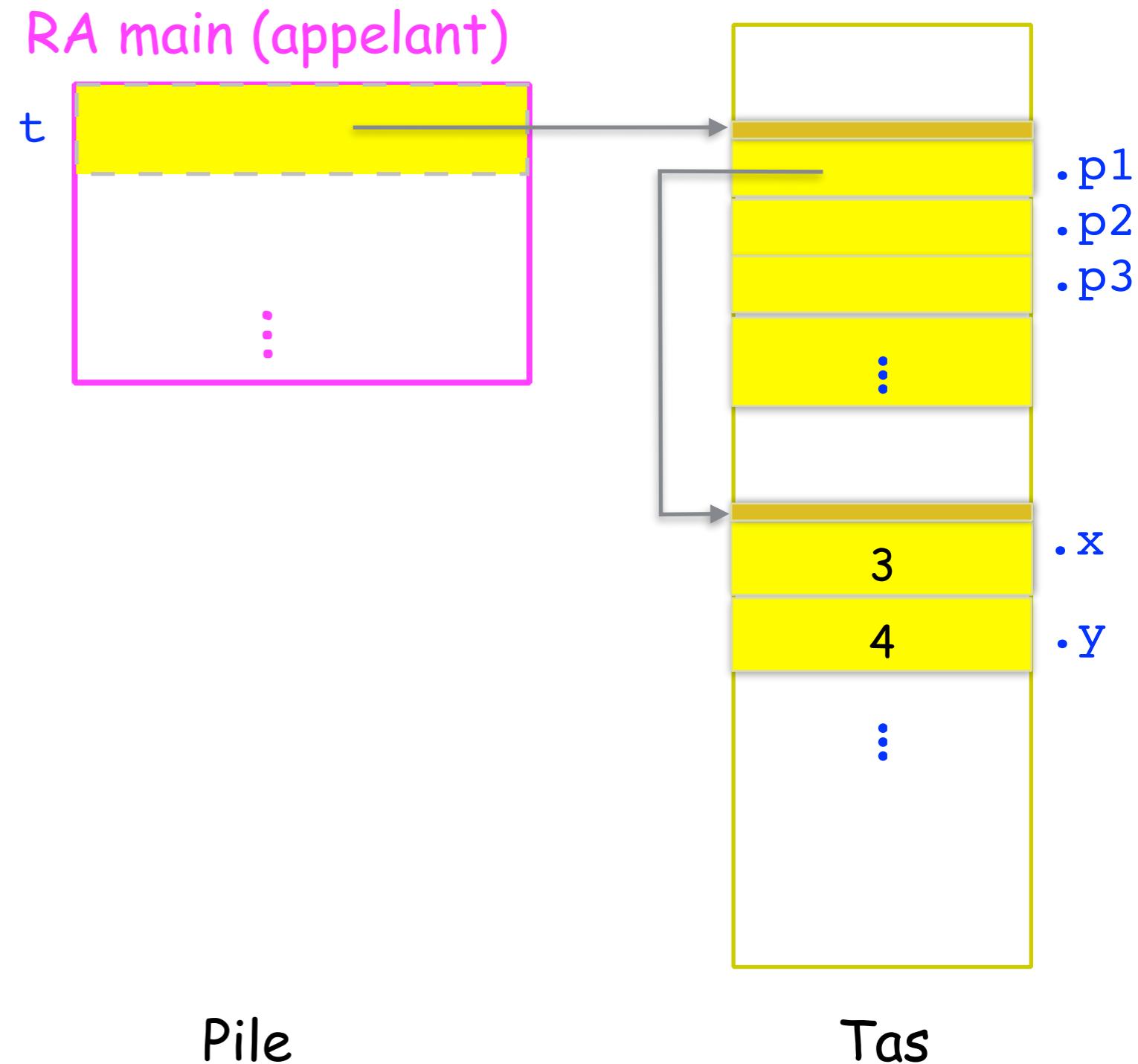
- Supposons de vouloir ajouter des méthodes d'accès à la classe Triangle qui permettent de connaître ses sommets (en lecture seule)
- La solution suivante n'est pas bonne :

```
public class Triangle {  
    private Point p1, p2, p3; //sommets  
    ...  
    public getP1() { return p1; }  
    public getP2() { return p2; }  
    public getP3() { return p3; }  
    ...  
}
```

- Puisque p1, p2 p3 sont des références, les points du triangle sont modifiables par l'appelant
- Encapsulation violée

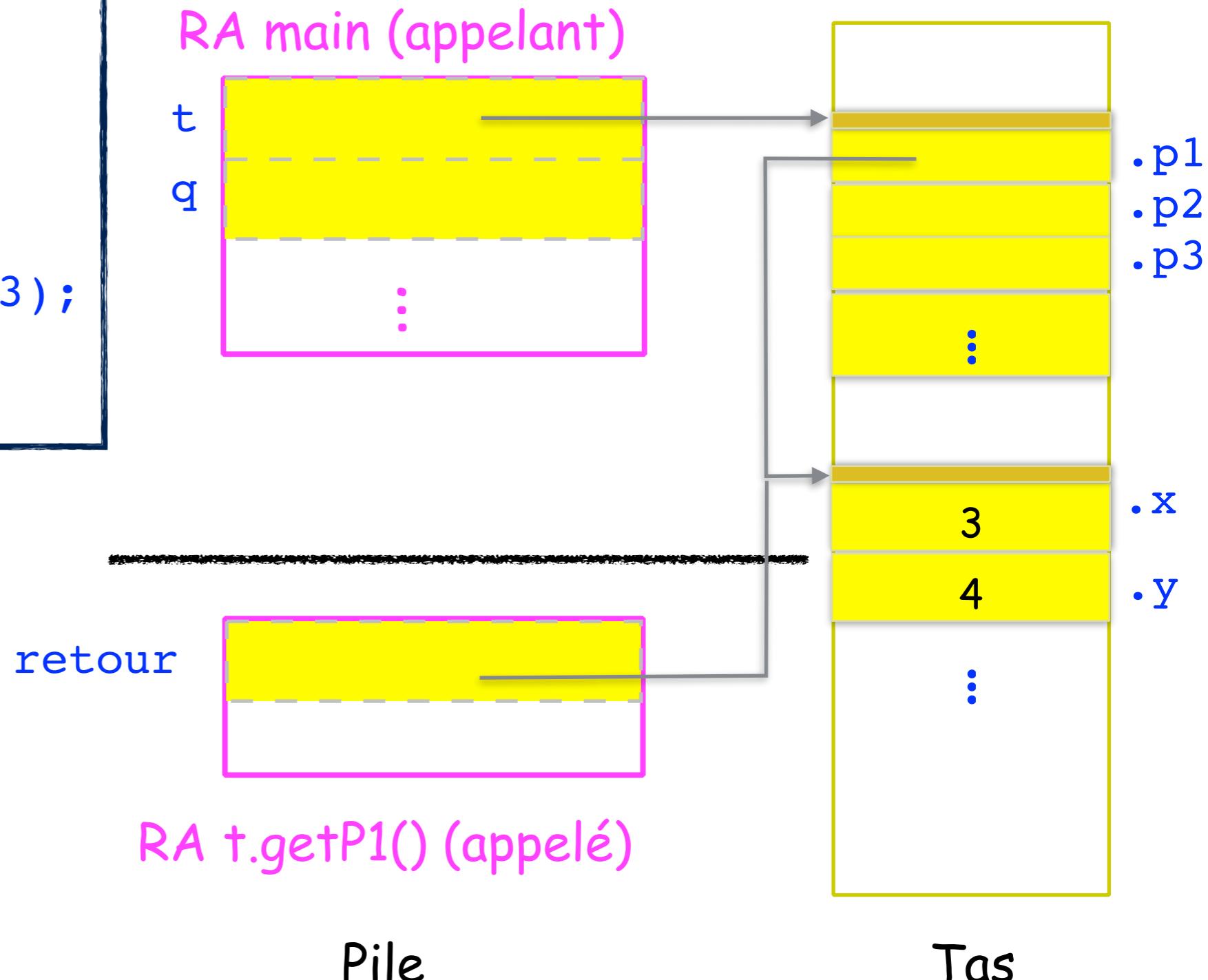
Préserver l'encapsulation

```
//main  
Point q1 =  
new Point (3,4);  
Point q2 = ...;  
Point q3 = ...;  
Triangle t =  
new Triangle(q1,q2,q3);
```



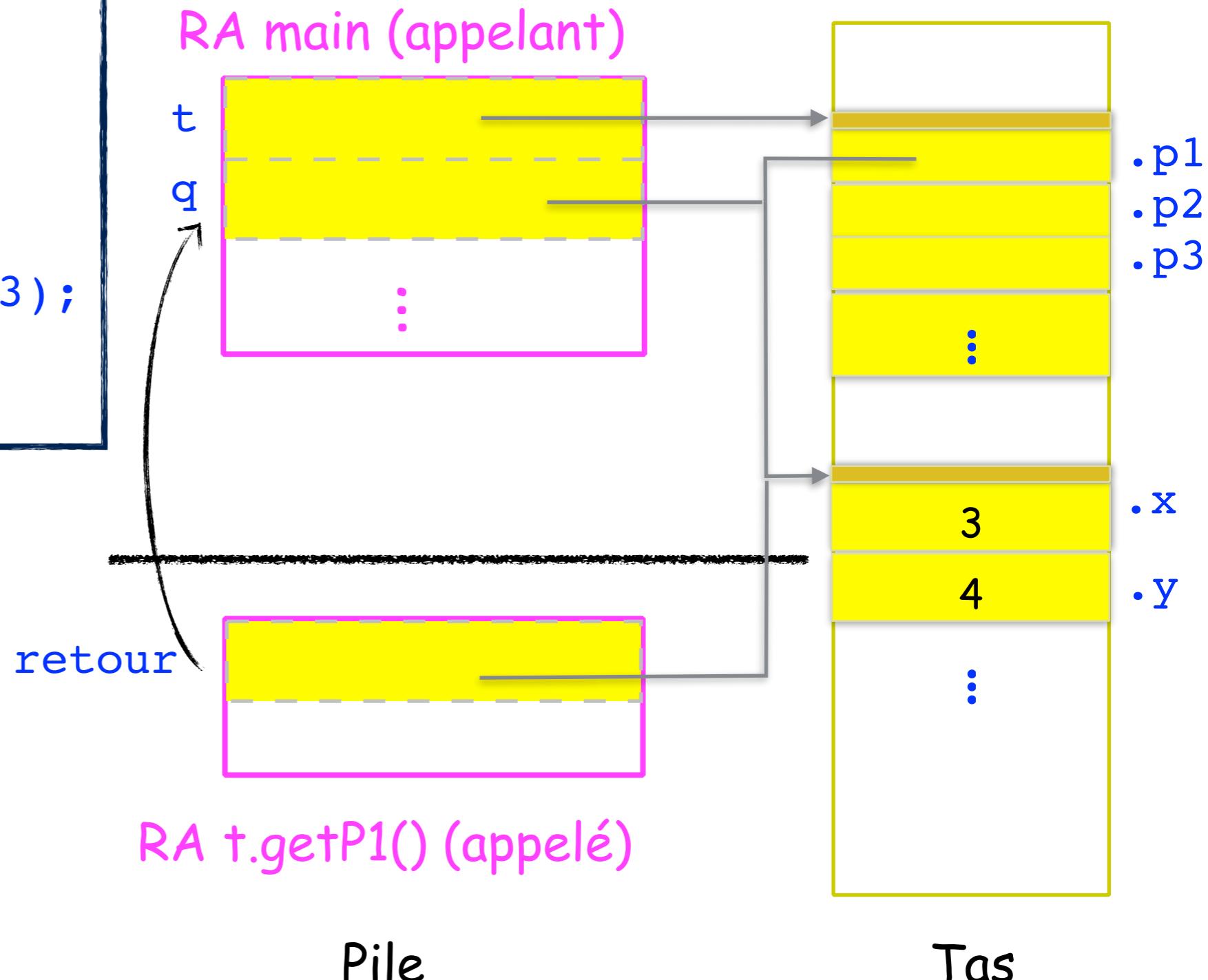
Préserver l'encapsulation

```
//main  
Point q1 =  
new Point (3,4);  
Point q2 = ...;  
Point q3 = ...;  
Triangle t =  
new Triangle(q1,q2,q3);  
Point q = t.getP1();
```



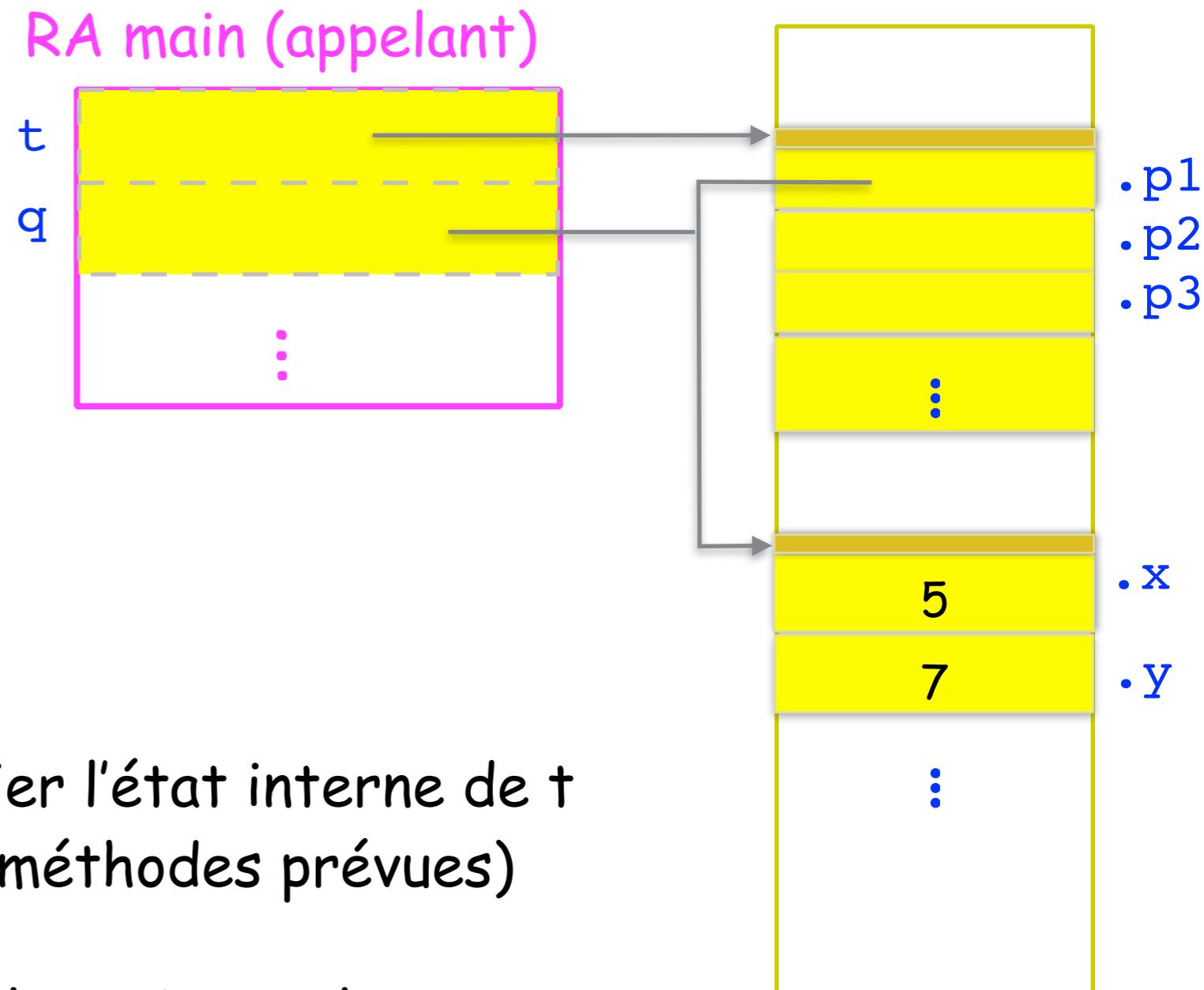
Préserver l'encapsulation

```
//main  
Point q1 =  
new Point (3,4);  
Point q2 = ...;  
Point q3 = ...;  
Triangle t =  
new Triangle(q1,q2,q3);  
Point q = t.getP1();
```



Préserver l'encapsulation

```
//main  
Point q1 =  
new Point (3,4);  
Point q2 = ...;  
Point q3 = ...;  
Triangle t =  
new Triangle(q1,q2,q3);  
Point q = t.getP1();  
q.deplace(5,7);
```



Pas OK l'appelant peut modifier l'état interne de *t*
(de façon autre que avec les méthodes prévues)

Encapsulation violée pour la classe Triangle

Tas

Préserver l'encapsulation

Ce qu'il faut retenir

Méthodes d'accès publiques : éviter de retourner une référence à un champ privé qui soit un objet mutable

- Classe mutable : classe possédant des méthodes qui modifient l'état interne (e.g Point, Array, ...)
- Exemples de classes immuables :
 - String
 - LocalDate
 - ...

Préserver l'encapsulation

Méthodes d'accès publiques : éviter de retourner une référence à un champ privé qui soit un objet mutable

Comment garantir l'accès en lecture à ces champs?

Clonage

Préserver l'encapsulation : Clonage

les méthodes d'accès retourner normalement une copie des objets mutables

```
public class Triangle {  
    private Point p1, p2, p3; //sommets  
  
    ...  
    public getP1() { return new Point(p1); }  
    public getP2() { return new Point(p2); }  
    public getP3() { return new Point(p3); }  
  
    ...  
}
```

Préserver l'encapsulation : exceptions

- La règle précédente a des exceptions :
- parfois on souhaite explicitement donner accès à une partie de l'état interne d'un objet avec droit de modification

Règles de visibilité des noms

Noms et Blocs

- **Nom** : un nom de variable, paramètre ou méthode
- **Bloc** : portion de code entourée par { }
- Trois type de blocs
 1. bloc de **classe**

```
Class A{
```

...

}

2. bloc de **méthode**

```
void f() {
```

...

}

3. Bloc délimitant une portion de **code** (if, boucle, initialisation, etc.) qui n'est pas des deux types précédents

- Les blocs de différents types peuvent être **imbriqués** l'un dans l'autre (mais pas arbitrairement)
 - niveau arbitrairement profond d'imbrication
- Chaque **nom** est introduit (déclaré) dans un bloc

Règles de visibilité des noms

- 1. Un nom introduit dans un **bloc de classe** est visible partout dans ce bloc et pénètre dans les blocs imbriqués de tout niveau

```
class A {  
    void f (){  
        ... x ... // le champ est visible même si  
                   // introduit plus loin  
        {  
            ...x...  
        }  
    }  
    private int x;  
}
```

Règles de visibilité des noms

- 2. Un nom introduit dans un **bloc de classe** peut être redéfini dans un bloc imbriqué dans celui-là (de tout niveau d'imbrication). Dans ce cas, dans le bloc imbriqué, le nom est caché par sa redéfinition

```
class A {  
    void f (){  
        ... x ...  
  
        { int x;  
            ...x... //Ne fait plus référence au champ  
        } //mais à la variable déclarée ci-dessus  
        ... x ... // Ici x fait de nouveau référence au champ  
    }  
    private int x;  
}
```

Règles de visibilité des noms

3. Un nom introduit dans un **bloc qui n'est pas de classe** est visible dans le bloc, y compris ses blocs imbriqués, mais uniquement à partir de son introduction. Il n'est pas visible en dehors du bloc.

```
public int f(int M) {  
    int n = 0;  
    int sum = 0;  
    do {  
        x = 0; //ERREUR  
        int x = 1;  
        for (int i =1; i<n; i++) {  
            x = x*i;  
        }  
        i = 0; //ERREUR  
        sum = sum + x  
        n = n+1;  
    } while (sum < M)  
    return n;  
}
```

Règles de visibilité des noms

4. Un nom introduit dans un **bloc qui n'est pas de classe** ne peut pas être redéfini **plus loin** dans le même bloc (même pas dans un de ses blocs imbriqués)

```
public int f(int M) {  
    int n = 0;  
    int sum = 0;  
    do {  
        double n = 0; //ERREUR  
        int x = 1;  
        for (int i =1; i<n; i++) {  
            x = x*i;  
        }  
        sum = sum + x  
        n = n+1;  
    } while (sum < M)  
    int n = 0; //ERREUR  
    return n;  
}
```