

## Object-Oriented Programming

### First Checkpoint

Pr. Olivier Gruber

([olivier.gruber@univ-grenoble-alpes.fr](mailto:olivier.gruber@univ-grenoble-alpes.fr))

Laboratoire d'Informatique de Grenoble

Université de Grenoble-Alpes

- Un retour sur les concepts
- S'appropriier les concepts

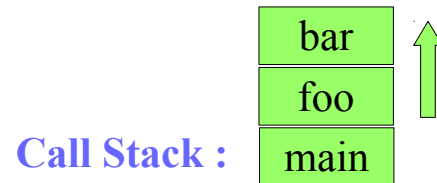
- Des blocs de code : { ... }
  - Des variables locales
  - Une séquence de « statements »
- Statements
  - Des expressions pour invoquer des fonctions/méthodes et calculer des valeurs
  - Des assignations pour assigner des valeurs aux variables
  - Des structures de contrôle du flot d'exécution (if-then-else, while, switch)
- Des fonctions
  - Avec des arguments
  - Un bloc de code
  - Une valeur de retour ou *void*

- Noms symboliques
  - Pour les variables et arguments
  - Nomme une zone mémoire qui est typée
- Types
  - Nom symbolique associé à un type
  - Le type contraint la valeur que peut contenir la zone mémoire
- Valeurs possibles
  - Primitives : int, float, char, ...
  - Référence : identité d'un objet

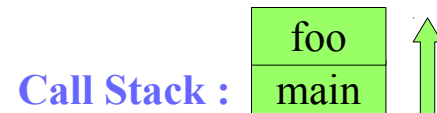
# La Pile d'Exécution

5

- Notion de pile d'exécution
  - Copie pour les arguments



→ `int32_t bar(int32_t arg) {` ●  
`int32_t val = 3 * arg;`  
`return val;`  
`}`



→ `int32_t foo(int32_t arg) {` ●  
`int32_t val = 2 * bar(arg);` ●  
`return val;`  
`}`

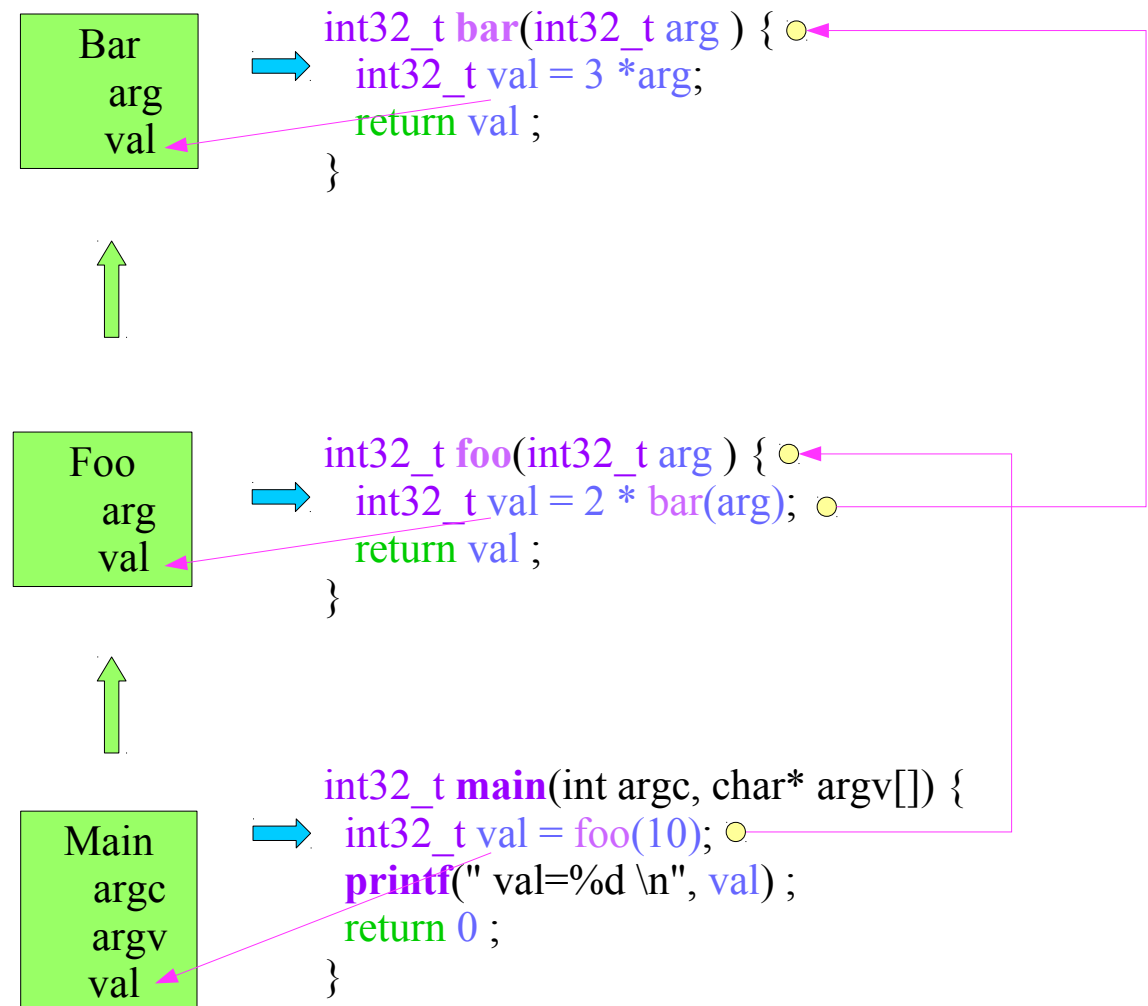


→ `int32_t main(int argc, char* argv[]) {`  
`int32_t val = foo(10);` ●  
`printf(" val=%d \n", val);`  
`return 0;`  
`}`

# La Pile d'Exécution

6

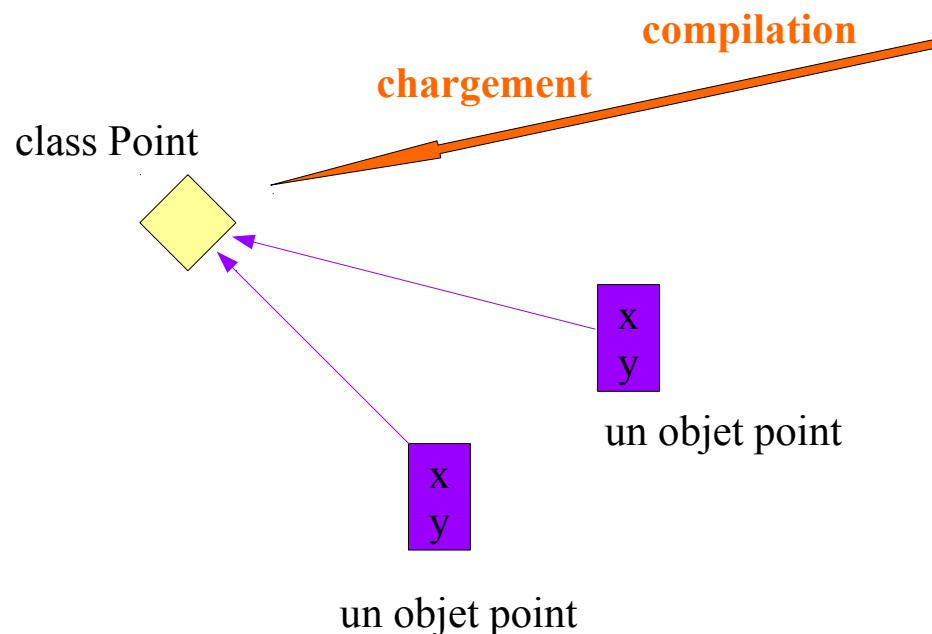
- Notion de visibilité depuis un site dans le code
  - A l'exécution, nom symbolique → zone mémoire



# Le Paradigme Orienté Objet

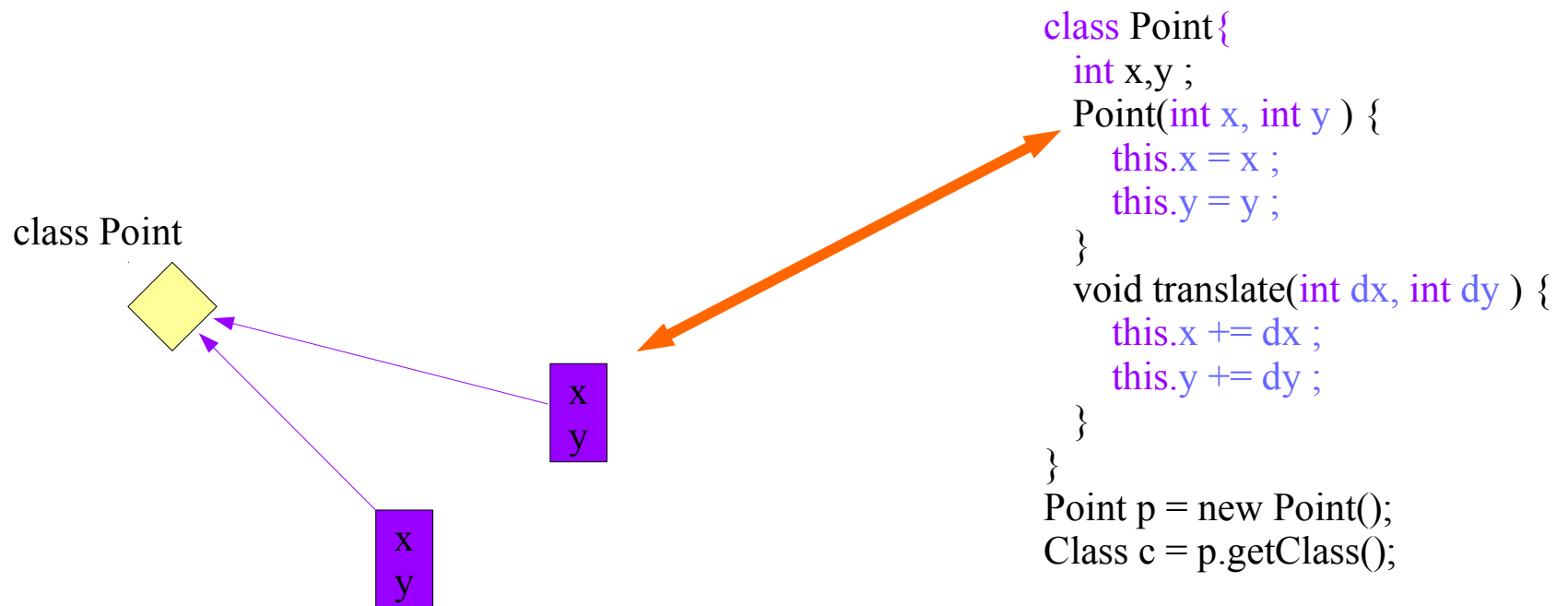
7

- Chaque classe décrit ses instances
  - En termes de structure avec des champs – noms symboliques et types
  - En termes de comportement avec des méthodes – signatures
- Chaque objet est une instance d'une classe
  - Chaque objet connaît sa classe
  - Cette classe sert d'assistant à l'exécution



```
class Point{  
    int x,y ;  
    Point(int x, int y ) {  
        this.x = x ;  
        this.y = y ;  
    }  
    void translate(int dx, int dy ) {  
        this.x += dx ;  
        this.y += dy ;  
    }  
}  
Point p = new Point();  
Class c = p.getClass();
```

- Une classe est une fabrique d'objets
  - Un objet connaît sa classe
  - Un objet correspondent à la description de sa classe
  - Un nouvel objet est « *construit* » par un constructeur





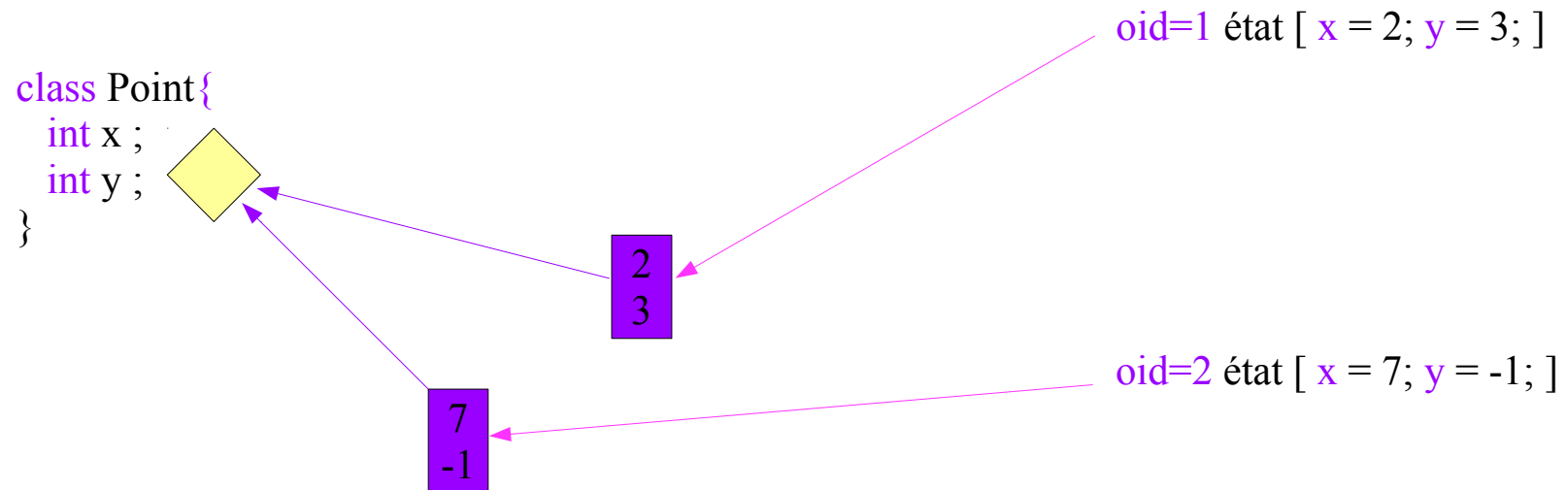
- **Objects**

- Des « individus »

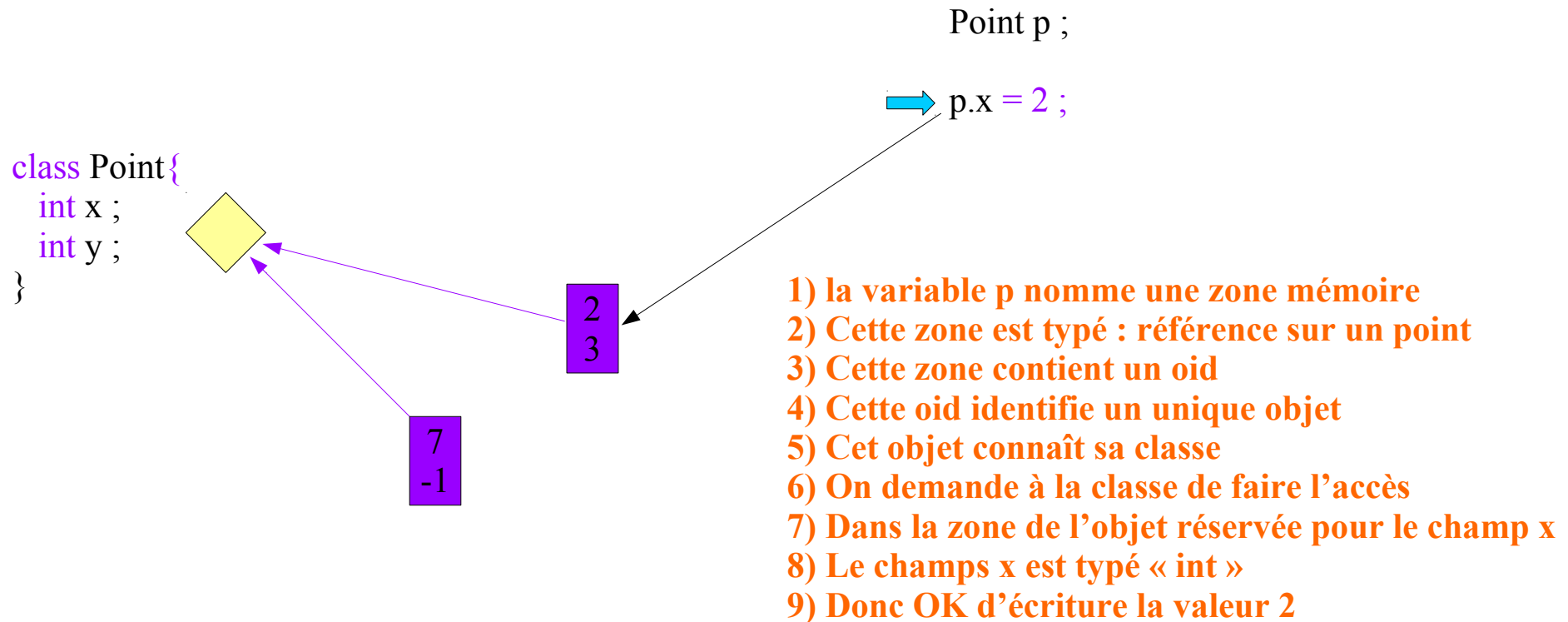
- Uniquement identifié (object identifier ~ oid)
    - Les oids sont les valeurs des références (variables, arguments et champs)

- Chaque objet a un état

- Etat = les valeurs que contiennent ses champs

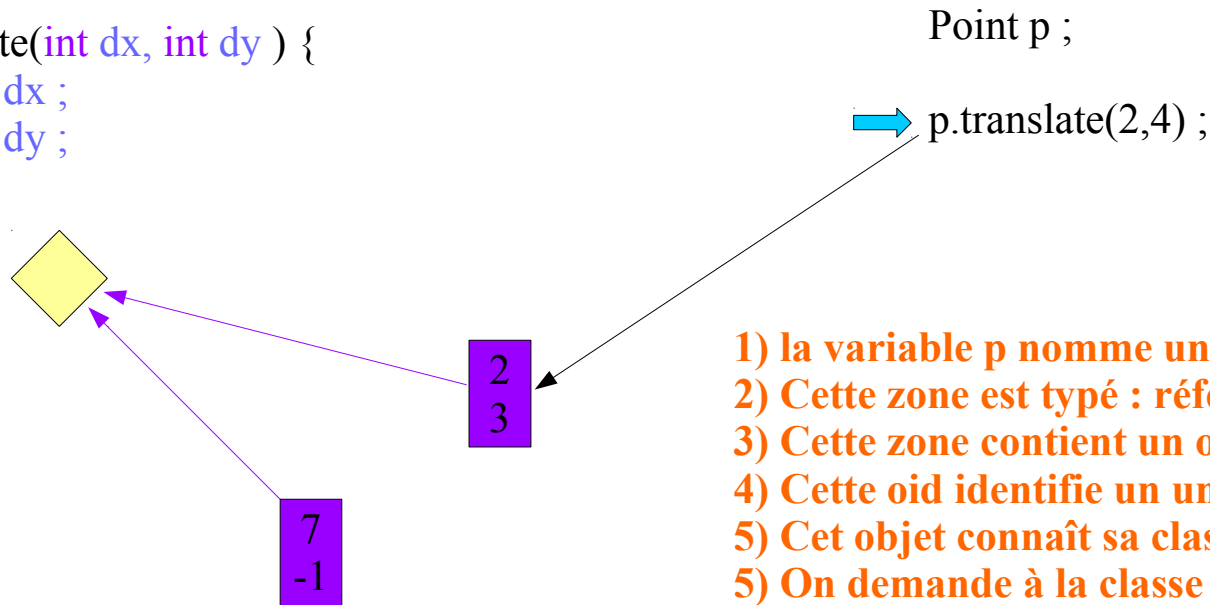


- Accéder à un champ d'un objet
  - Il faut un objet et un nom de champ
  - La classe vérifie que le champ existe
  - La classe fait la lecture ou l'écriture du champ



- Invoquer une méthode sur un objet
  - Il faut un objet et une signature de méthode
  - La classe trouve la méthode qui correspond à la signature

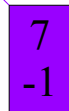
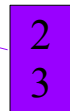
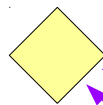
```
class Point{  
  int x ;  
  int y ;  
  void translate(int dx, int dy ) {  
    this.x += dx ;  
    this.y += dy ;  
  }  
}
```



- 1) la variable p nomme une zone mémoire
  - 2) Cette zone est typé : référence sur un point
  - 3) Cette zone contient un oid
  - 4) Cette oid identifie un unique objet
  - 5) Cet objet connaît sa classe
- 5) On demande à la classe la méthode :  
**void translate(int, int)**

- La surcharge de méthode
  - Même nom, mais une signature différente

```
class Point {  
    int x ;  
    int y ;  
    void translate(Point p) {  
        this.x += p.x ;  
        this.y += p.y ;  
    }  
    void translate(int dx, int dy) {  
        this.x += dx ;  
        this.y += dy ;  
    }  
}
```



Point p,q ;

→ p.translate(2,4) ;

p.translate(q) ;

Deux signatures différentes :

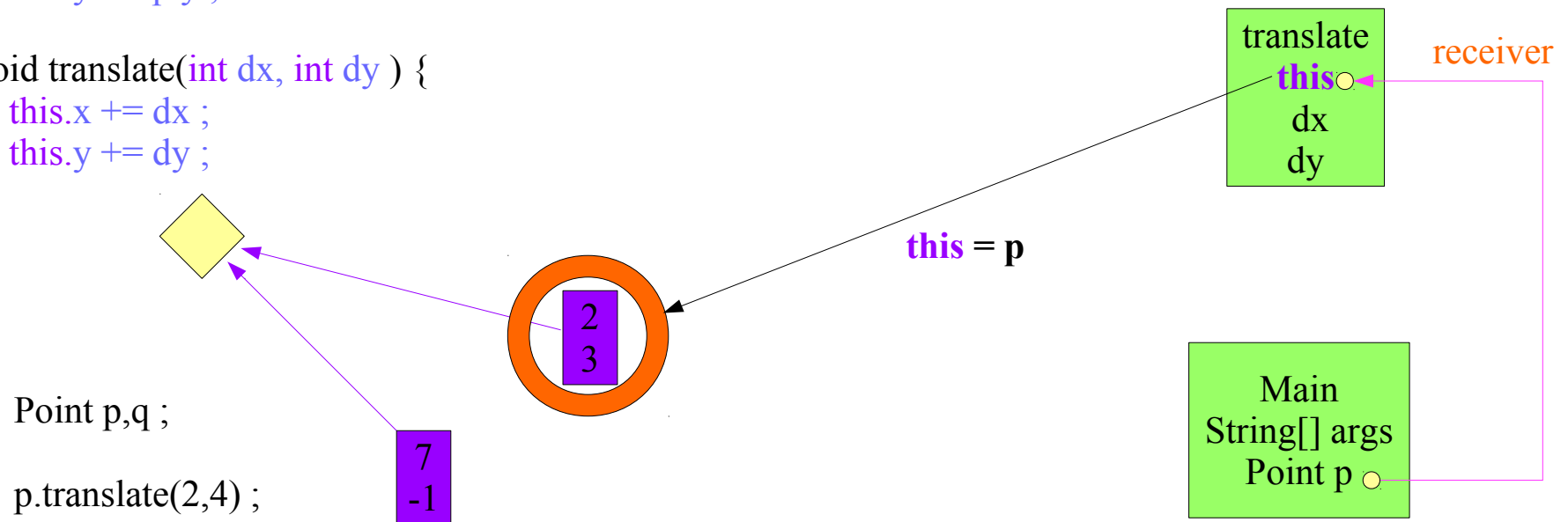
→ void translate(int, int)

void translate(Point p)

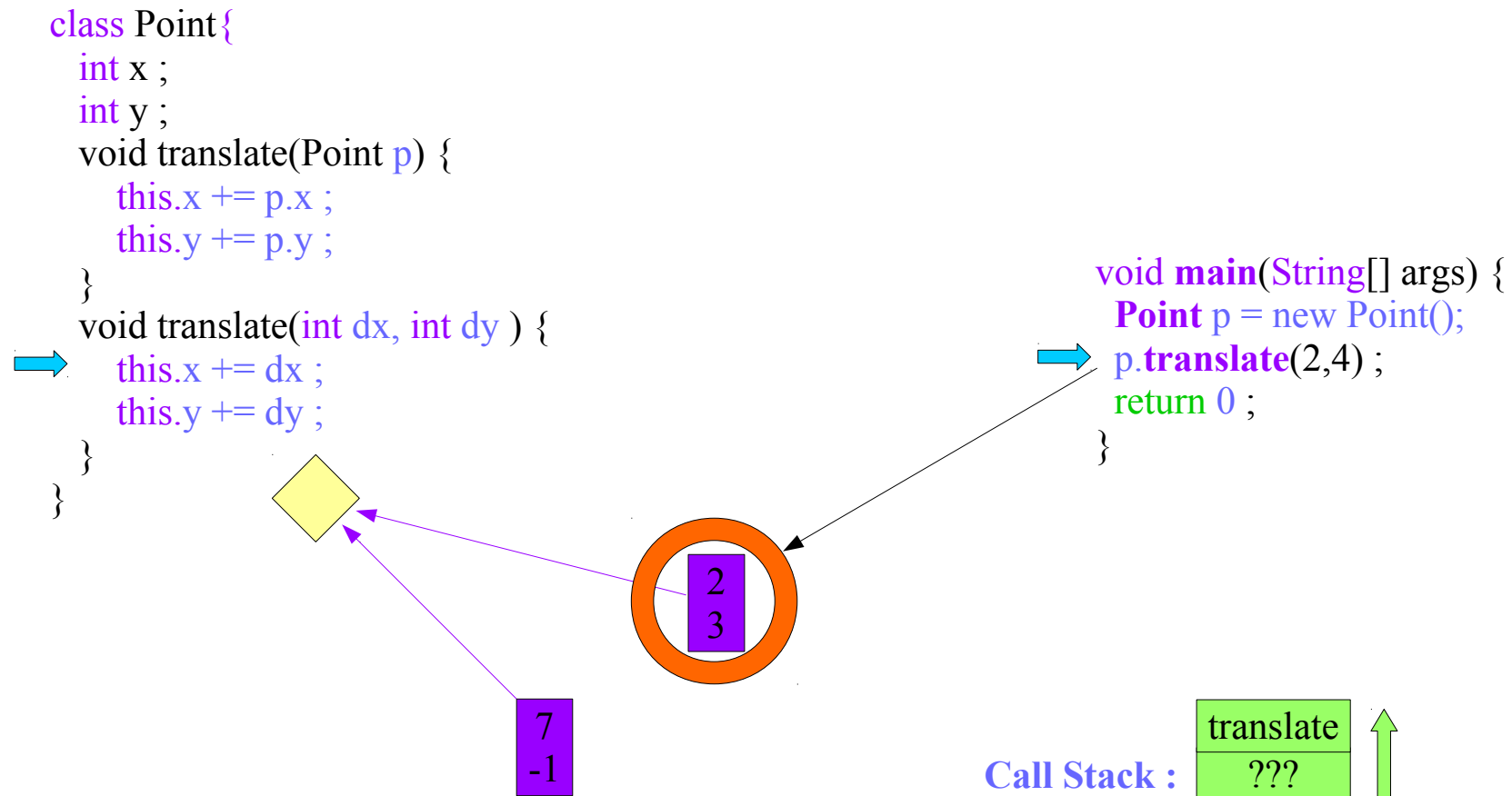
- Le receveur « this »
  - C'est l'objet sur lequel on invoque une méthode

```
class Point {  
    int x ;  
    int y ;  
    void translate(Point p) {  
        this.x += p.x ;  
        this.y += p.y ;  
    }  
    void translate(int dx, int dy ) {  
        this.x += dx ;  
        this.y += dy ;  
    }  
}
```

```
void main(String[] args) {  
    Point p = new Point();  
    p.translate(2,4) ;  
    return 0 ;  
}
```



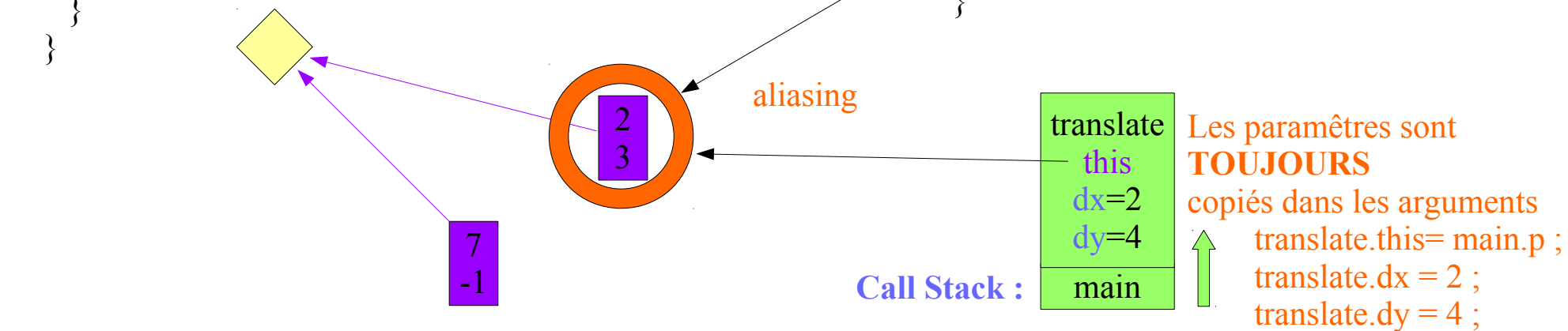
- L'invocation se met en place
  - Empile un nouveau contexte sur la pile d'exécution



- L'invocation se met en place
  - Empile un nouveau contexte sur la pile d'exécution

```
class Point {  
    int x ;  
    int y ;  
    void translate(Point p) {  
        this.x += p.x ;  
        this.y += p.y ;  
    }  
    void translate(int dx, int dy ) {  
        this.x += dx ;  
        this.y += dy ;  
    }  
}
```

```
void main(String[] args) {  
    Point p = new Point();  
    p.translate(2,4) ;  
    return 0 ;  
}
```

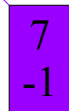
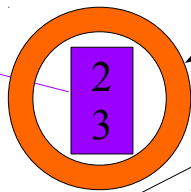
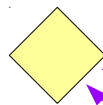


- L'invocation se met en place
  - Empile un nouveau contexte sur la pile d'exécution

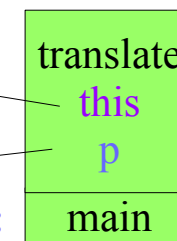
```
class Point{  
    int x ;  
    int y ;  
    void translate(Point p) {  
        → this.x += p.x ;  
        this.y += p.y ;  
    }  
    void translate(int dx, int dy ) {  
        this.x += dx ;  
        this.y += dy ;  
    }  
}
```

```
void main(String[] args) {  
    Point p = new Point();  
    p.translate(2,4) ;  
}
```

```
Point q = new Point();  
→ p.translate(q) ;  
return 0 ;  
}
```



Call Stack :



Les paramètres sont  
**TOUJOURS**  
copiés dans les arguments  
translate.this = main.p ;  
translate.p = main.q ;



# Classes – Objets – Exécution

17

- Il nous manque l'exécution
  - Le cuisinier qui se promène d'objet en objet
  - Il cuisine quelque chose en suivant les recettes des méthodes

```
class Point{  
  int x ;  
  int y ;  
  void translate(Point p) {  
    this.x += p.x ;  
    this.y += p.y ;  
  }  
  void translate(int dx, int dy ) {  
    this.x += dx ;  
    this.y += dy ;  
  }  
}
```



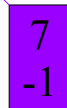
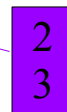
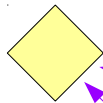
method



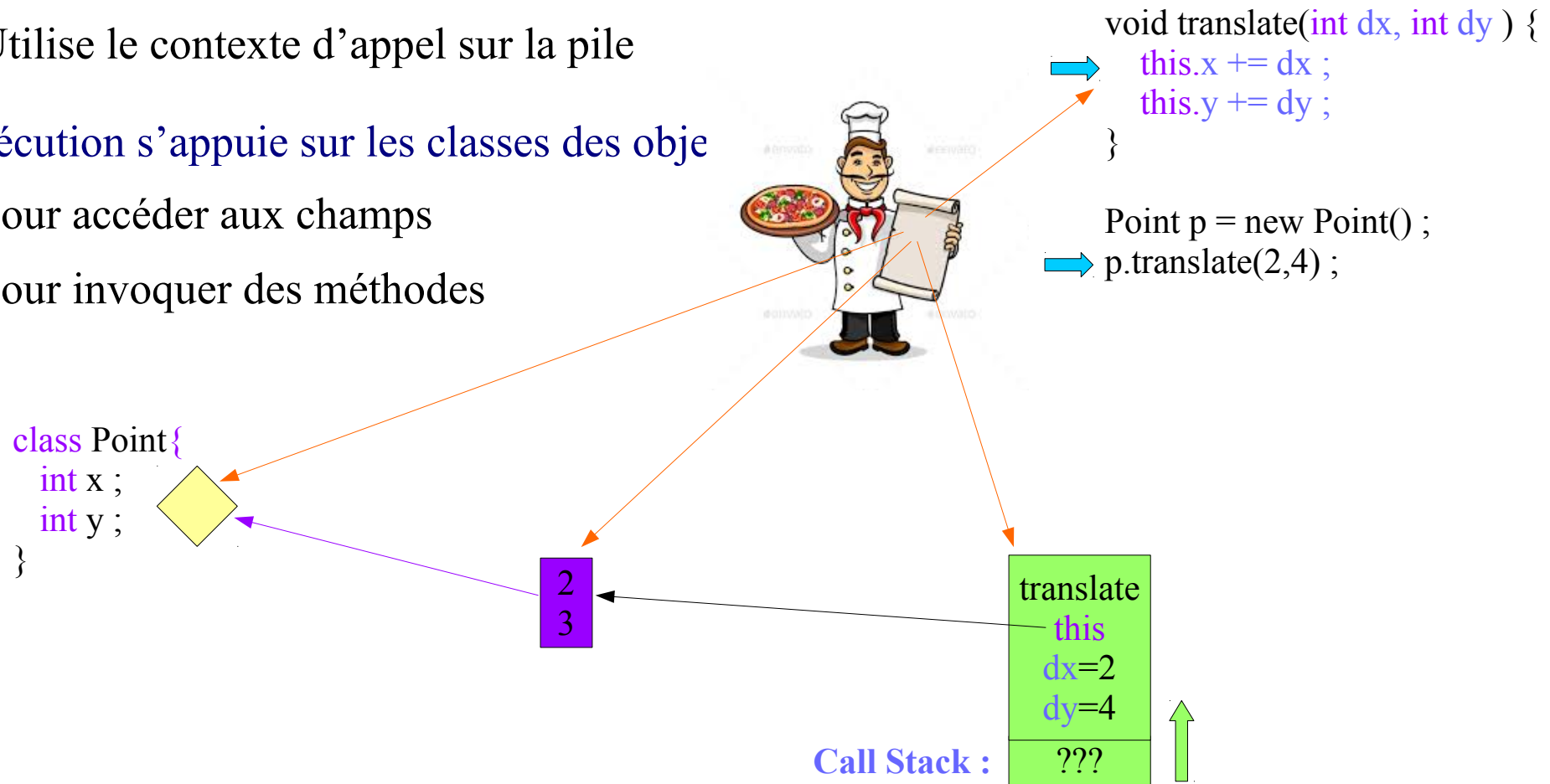
The cuisinier exécute la recette

cook = processor  
recette = méthode

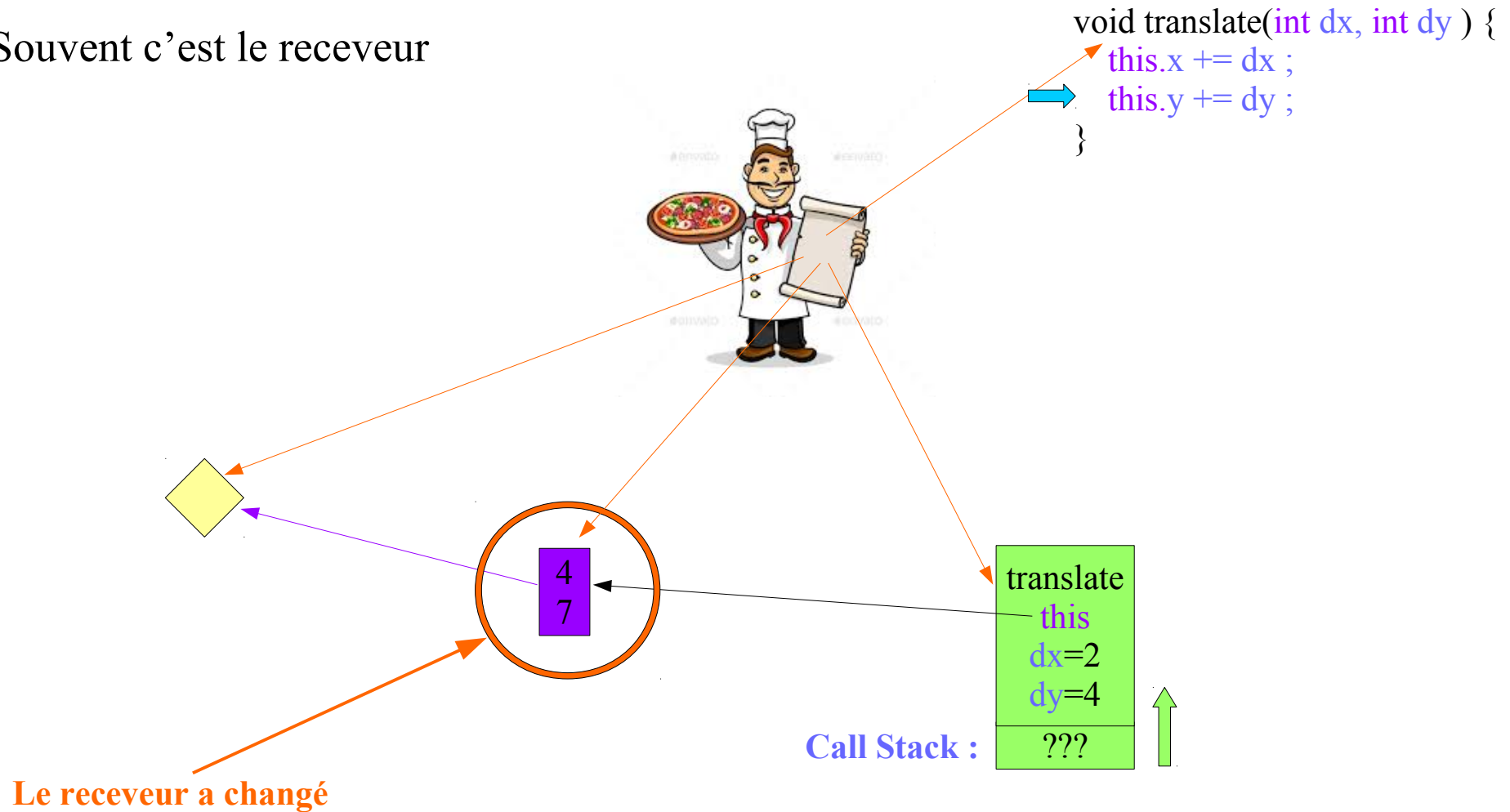
receiver



- Exécution de la méthode
  - Exécute les instructions
  - Utilise le contexte d'appel sur la pile
- Exécution s'appuie sur les classes des objets
  - Pour accéder aux champs
  - Pour invoquer des méthodes



- Quels sont les effets de l'exécution ?
  - Quels objets sont modifiés ?
  - Souvent c'est le receveur



- Exécution de la méthode

- Quels objets seront modifiés ?

```
Vector v = new Vector(4,PI/8);
```

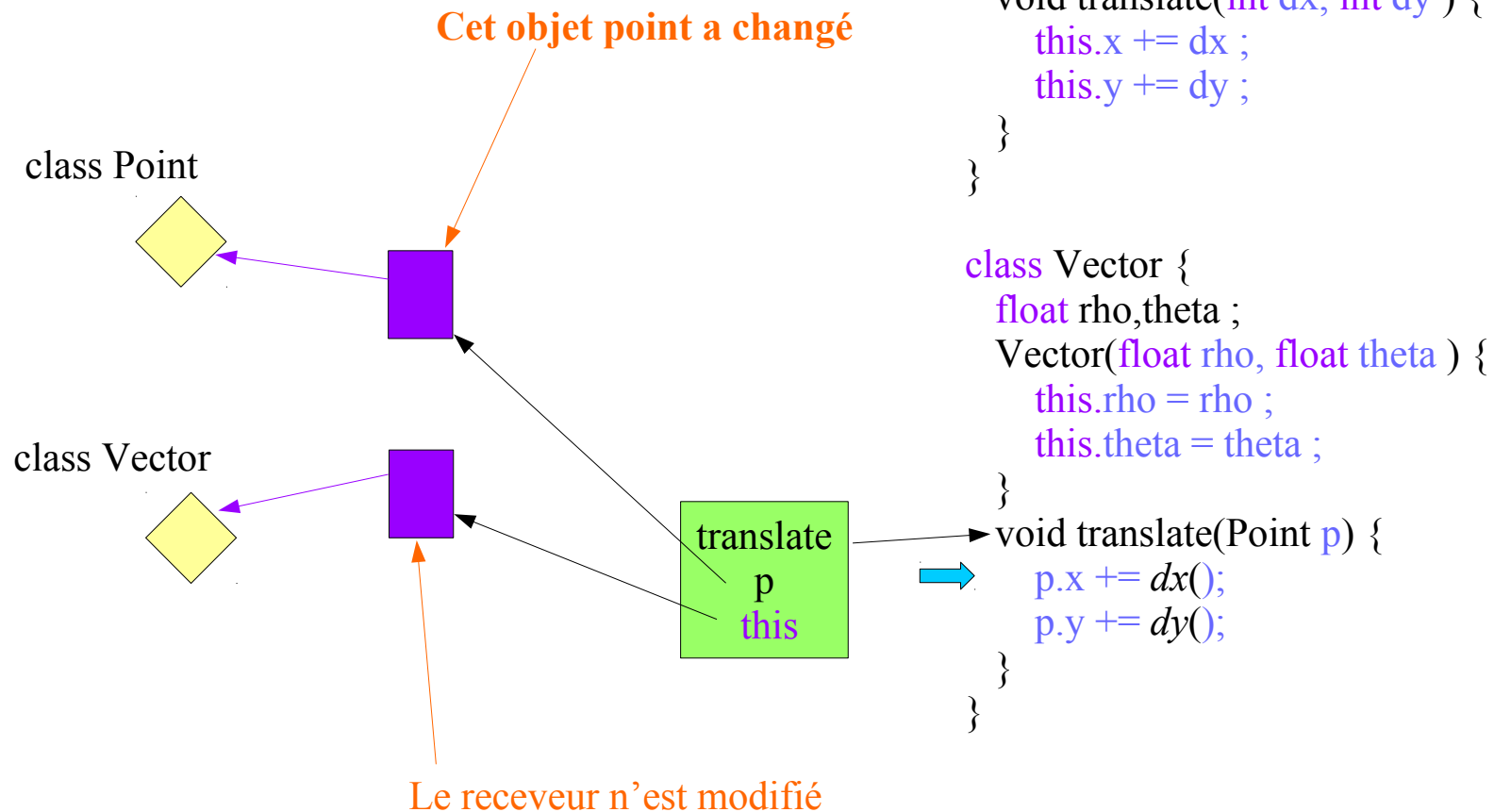
```
Point p = new Point(2,3);
```

```
➡ v.translate(p);
```

```
class Point{  
    int x,y ;  
    Point(int x, int y ) {  
        this.x = x ;  
        this.y = y ;  
    }  
    void translate(int dx, int dy ) {  
        this.x += dx ;  
        this.y += dy ;  
    }  
}
```

```
class Vector {  
    float rho,theta ;  
    Vector(float rho, float theta ) {  
        this.rho = rho ;  
        this.theta = theta ;  
    }  
    void translate(Point p) {  
        p.x += dx();  
        p.y += dy();  
    }  
}
```

- Exécution de la méthode
  - Quels objets seront modifiés ?
  - **L'objet passé en argument**



- Exécution de la méthode
  - Quels objets seront modifiés ?

```
Vector v = new Vector(4,PI/8);
```

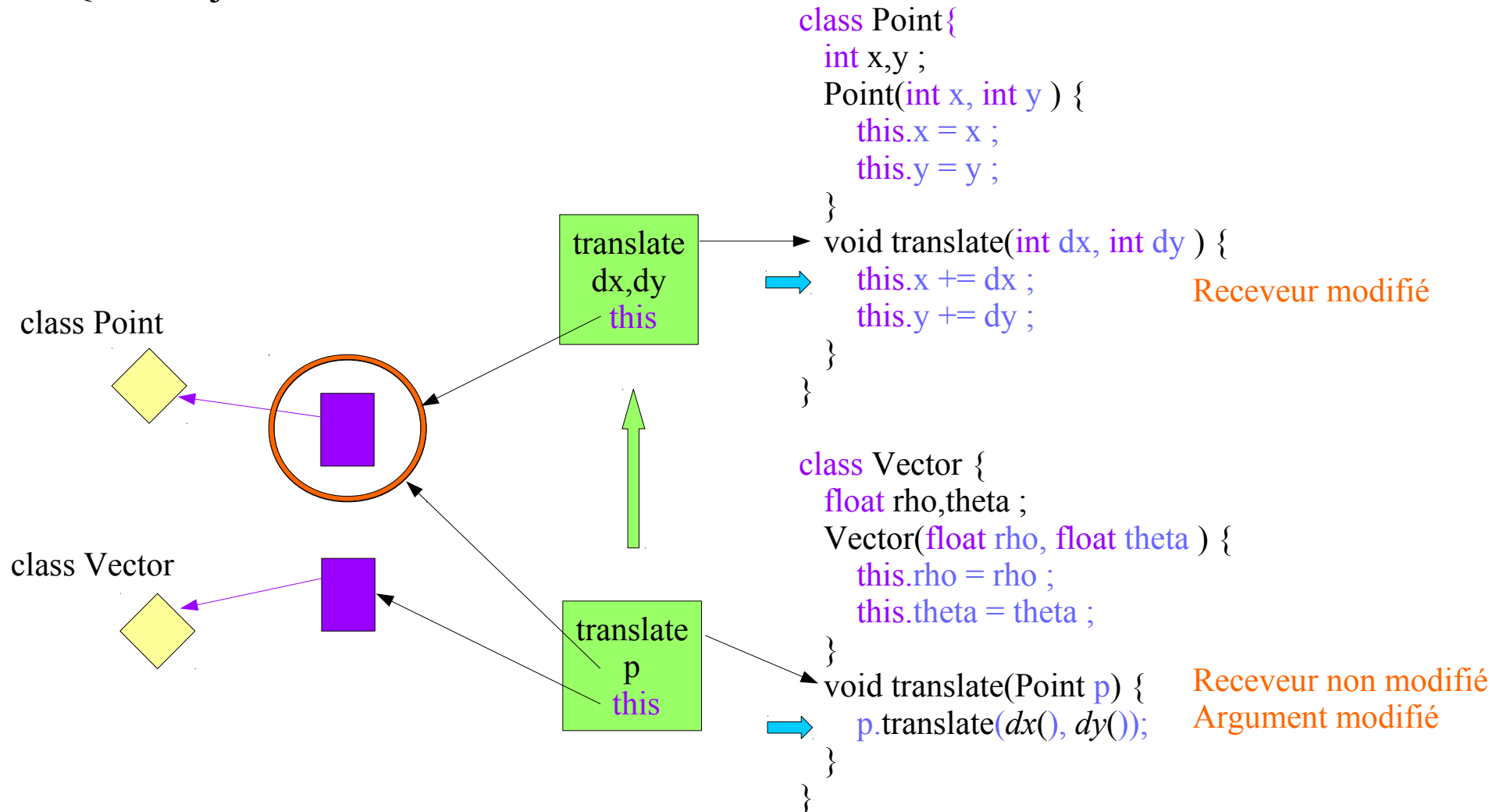
```
Point p = new Point(2,3);
```

```
➡ v.translate(p);
```

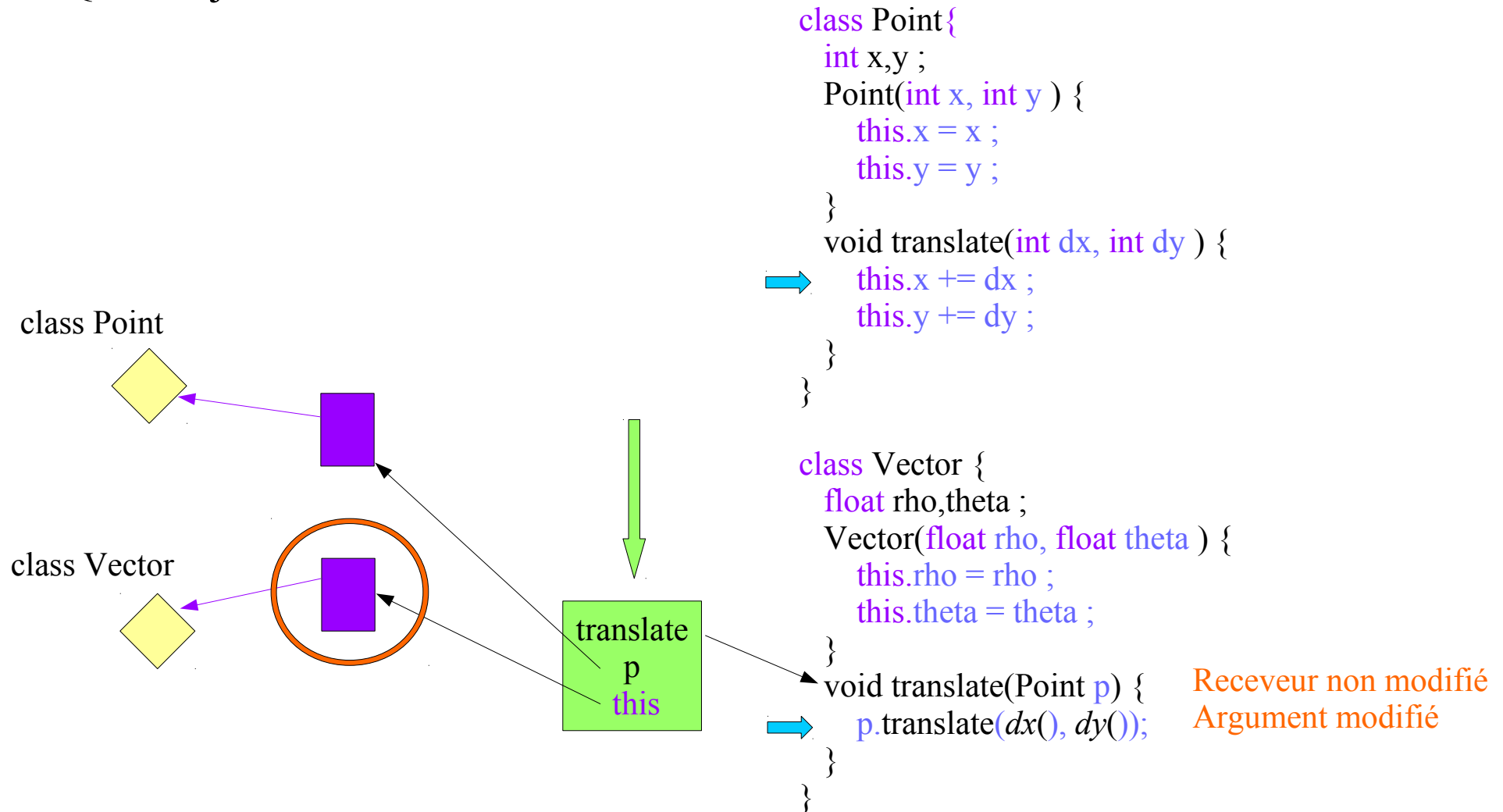
```
class Point{  
    int x,y;  
    Point(int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
    void translate(int dx, int dy) {  
        this.x += dx;  
        this.y += dy;  
    }  
}
```

```
class Vector {  
    float rho,theta;  
    Vector(float rho, float theta) {  
        this.rho = rho;  
        this.theta = theta;  
    }  
    void translate(Point p) {  
➡    p.translate(dx(), dy());  
    }  
}
```

- Exécution de la méthode
  - Quels objets seront modifiés ?



- Exécution de la méthode
  - Quels objets seront modifiés ?





- La visibilité
  - Qualifier : private vs public
  - Sur les champs et les méthodes
- La classe contrôle la visibilité
  - Depuis un site dans une méthode

- 1) la variable p nomme une zone mémoire
- 2) Cette zone contient un oid
- 3) Cette oid identifie un unique objet
- 4) Cet objet connaît sa classe
- 5) On demande à la classe :

Puis-je voir x ?  
Réponse : NON

```
class Point{  
    private int x,y ;  
    Point(int x, int y ) {  
        this.x = x ;  
        this.y = y ;  
    }  
    void translate(int dx, int dy ) {  
        this.x += dx ;  
        this.y += dy ;  
    }  
}
```

```
class Vector {  
    float rho,theta ;  
    Vector(float rho, float theta ) {  
        this.rho = rho ;  
        this.theta = theta ;  
    }  
    void translate(Point p) {  
        ➡ p.x += dx();  
        p.y += dy();  
    }  
}
```

- La visibilité
  - Qualifier : *private* ou *public*
  - Sur les champs et les méthodes
- La classe contrôle la visibilité
  - Depuis un site dans une méthode

- 1) la variable **p** nomme une zone mémoire
- 2) Cette zone contient un oid
- 3) Cette oid identifie un unique objet
- 4) Cet objet connaît sa classe
- 5) On demande à la classe :

Puis-je voir **void translate(int, int)** ?

Réponse : **NON**

```
class Point{  
    private int x,y ;  
    Point(int x, int y ) {  
        this.x = x ;  
        this.y = y ;  
    }  
    private void translate(int dx, int dy ) {  
        this.x += dx ;  
        this.y += dy ;  
    }  
}
```

```
class Vector {  
    float rho,theta ;  
    Vector(float rho, float theta ) {  
        this.rho = rho ;  
        this.theta = theta ;  
    }  
    void translate(Point p) {  
        → p.translate(dx(),dy()) ;  
    }  
}
```

- Doit-on écouter Eclipse ?

```
public class Point {  
    private int x,y ;  
    public Point(int x, int y ) {  
        this.x = x ;  
        this.y = y ;  
    }  
    public void translate(int dx, int dy ) {  
        this.x += dx ;  
        this.y += dy ;  
    }  
}
```

```
class Vector {  
    float rho,theta ;  
    Vector(float rho, float theta ) {  
        this.rho = rho ;  
        this.theta = theta ;  
    }  
    void translate(Point p) {  
        p.x += dx();  
        p.y += dy();  
    }  
}
```

**Eclipse propose deux solutions possibles :**

- 1) Passer en visibilité package ?
- 2) Définir des getters et setters ?




- Doit-on écouter Eclipse ?
  - Surement pas sans réfléchir !

Eclipse propose deux solutions possibles :

- 1) Passer en visibilité package ?
- 2) Définir des getters et setters ?

```
public class Point {  
    private int x,y ;  
    public Point(int x, int y ) {  
        this.x = x ;  
        this.y = y ;  
    }  
    public void translate(int dx, int dy ) {  
        this.x += dx ;  
        this.y += dy ;  
    }  
}
```

```
class Vector {  
    float rho,theta ;  
    Vector(float rho, float theta ) {  
        this.rho = rho ;  
        this.theta = theta ;  
    }  
    void translate(Point p) {  
        p.x += dx();  
        p.y += dy();  
    }  
}
```



- Doit-on définir des *setters* et *getters* ?
  - Pour les champs privés

```
public class Point{  
    private int x,y ;  
    public Point(int x, int y ) {  
        this.x = x ;  
        this.y = y ;  
    }  
    public void translate(int dx, int dy ) {  
        this.x += dx ;  
        this.y += dy ;  
    }  
    public int getX() { return this.x ; }  
    public void setX(int x) { this.x = x ; }  
  
    public int getY() { return this.y ; }  
    public void setY(int y) { this.y = y ; }  
}
```

- Doit-on définir des *setters* et *getters* ?
  - Pour les champs privés

Ils ne servent à rien ici

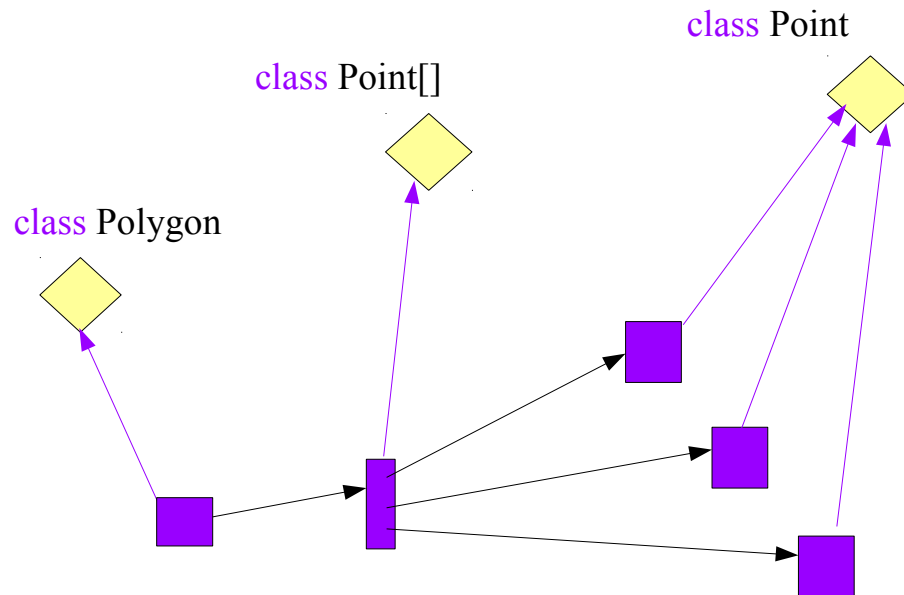
- 1) La méthode translate est suffisante
- 2) Et pourquoi ne pas avoir les champs en public ?

```
public class Point{  
    private int x,y ;  
    public Point(int x, int y ) {  
        this.x = x ;  
        this.y = y ;  
    }  
    public void translate(int dx, int dy ) {  
        this.x += dx ;  
        this.y += dy ;  
    }  
    public int getX(){ return this.x ; }  
public void setX(int x){ this.x = x ; }  
  
public int getY(){ return this.y ; }  
public void setY(int y){ this.y = y ; }  
}
```

- Encapsulation
  - Cacher les détails internes...
  - Permet de diviser le problème
  - De se partager le travail à faire
- Qu'est-ce qu'un polygone ?

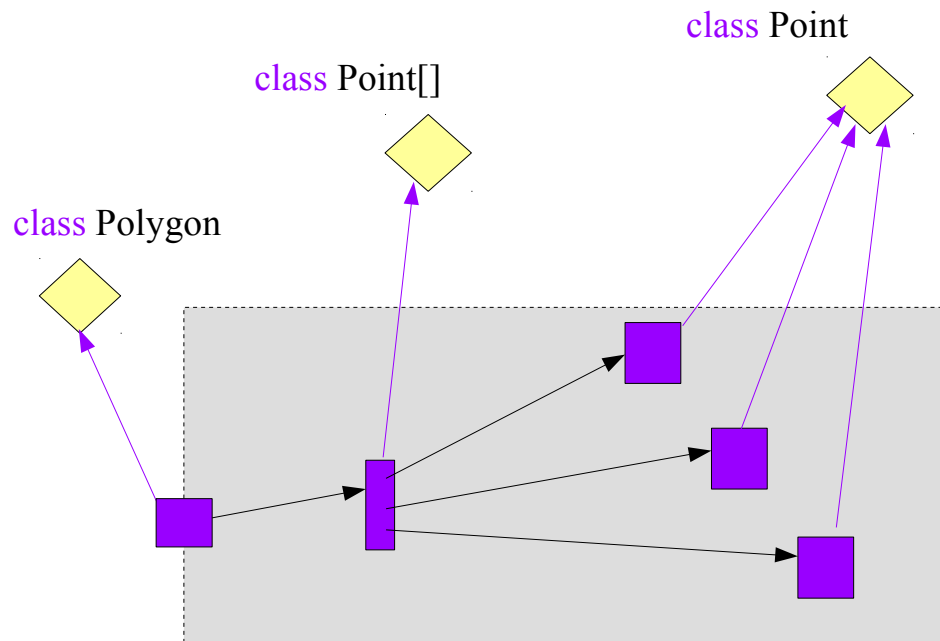
```
public class Point{  
    public int x,y ;  
    public Point(int x, int y ) {  
        this.x = x ;  
        this.y = y ;  
    }  
    public void translate(int dx, int dy ) {  
        this.x += dx ;  
        this.y += dy ;  
    }  
}
```

```
public class Polygon{  
    public Point points[];  
    public Polygon(Point[] pts ) {  
        this.points = pts ;  
    }  
}
```



- Encapsulation

- Vous pouvez ignorer les classes
- C'est sur le graphe d'objets



```
public class Point{  
    public int x,y ;  
    public Point(int x, int y ) {  
        this.x = x ;  
        this.y = y ;  
    }  
    public void translate(int dx, int dy ) {  
        this.x += dx ;  
        this.y += dy ;  
    }  
}
```

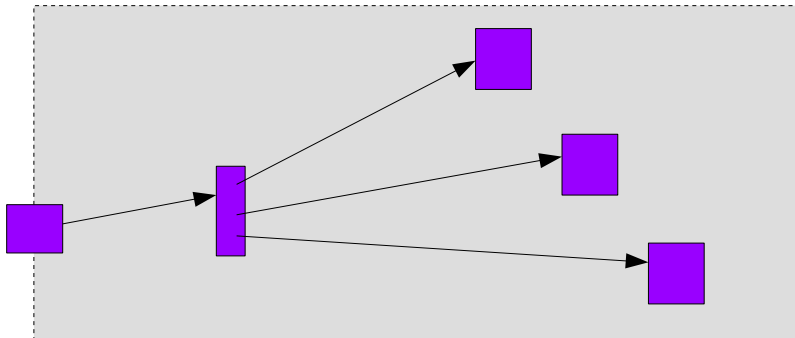
```
public class Polygon{  
    public Point points[];  
    public Polygon(Point[] pts ) {  
        this.points = pts ;  
    }  
}
```



- Encapsulation
  - Vous pouvez ignorer les classes
  - C'est sur le graphe d'objets
- Quels objets puis-je voir ?
  - **Tous les objets si tous les champs sont public**

Polygon p ;

p.points[2].x = 3 ;



```
public class Point{  
    public int x,y ;  
    public Point(int x, int y ) {  
        this.x = x ;  
        this.y = y ;  
    }  
    public void translate(int dx, int dy ) {  
        this.x += dx ;  
        this.y += dy ;  
    }  
}
```

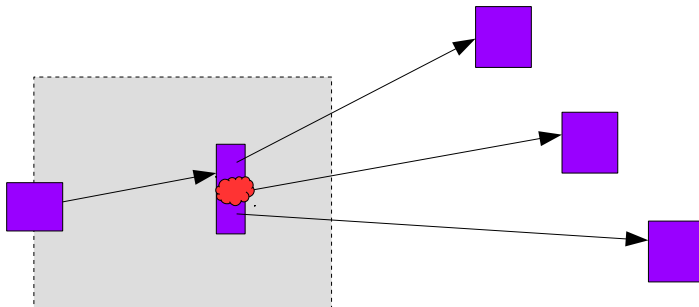
```
public class Polygon{  
    public Point points[];  
    public Polygon(Point[] pts ) {  
        this.points = pts ;  
    }  
}
```

- Encapsulation
  - C'est sur le graphe d'objets
- Quels objets puis-je voir ?
  - Usage de *private* mais **aliasing** possible

```
Polygon poly ;  
Point[] pts ;
```

```
Poly = new Polygon(pts) ;
```

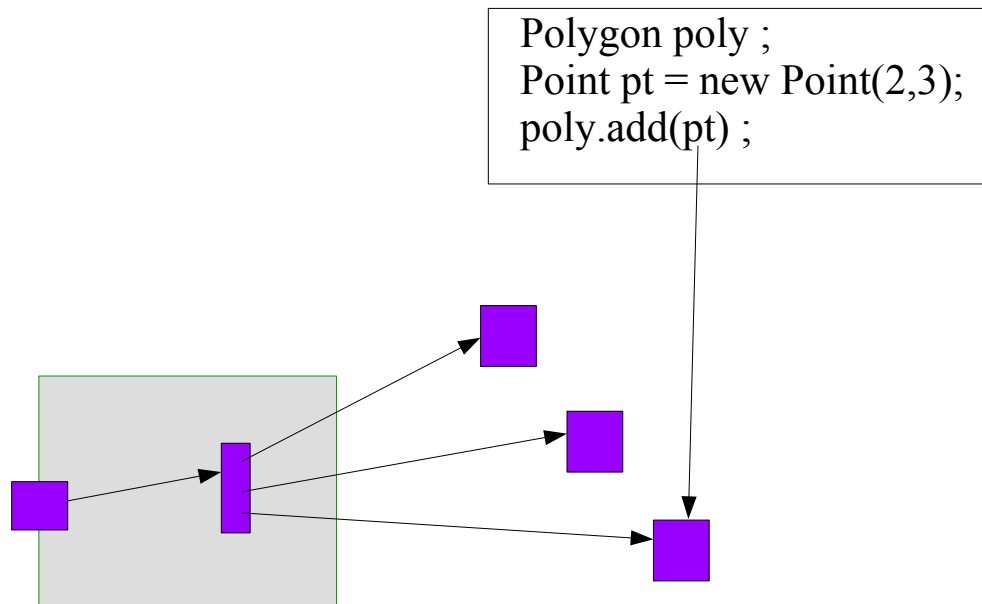
```
pts[2]=null ;
```



```
public class Point{  
    public int x,y ;  
    public Point(int x, int y ) {  
        this.x = x ;  
        this.y = y ;  
    }  
    public void translate(int dx, int dy ) {  
        this.x += dx ;  
        this.y += dy ;  
    }  
}
```

```
➡ public class Polygon{  
    private Point points[];  
    public Polygon(Point[] pts ) {  
        this.points = pts ;  
    }  
}
```

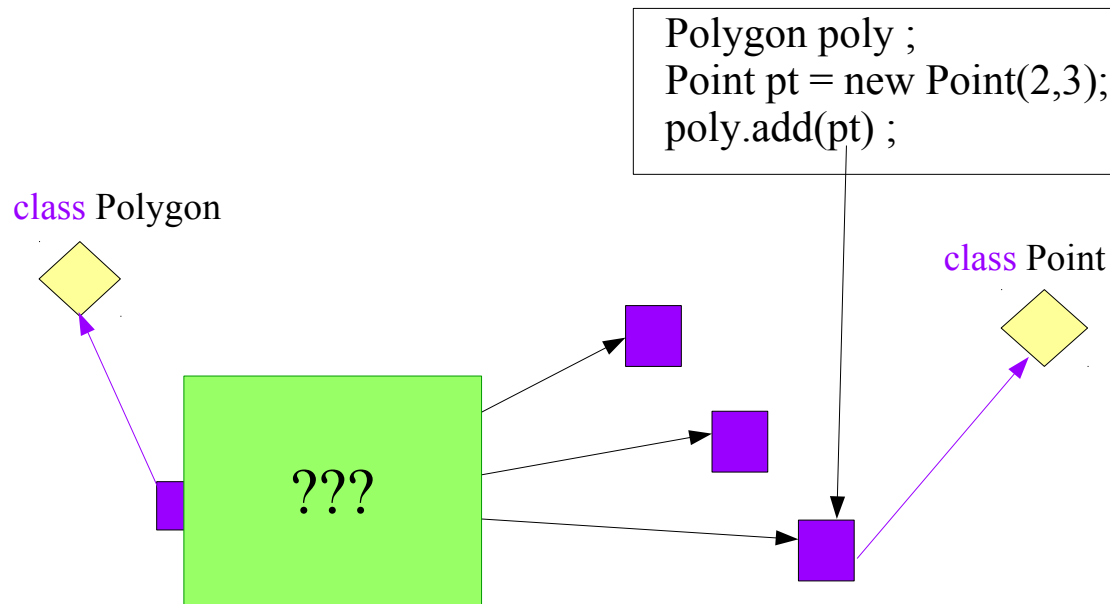
- Encapsulation
  - C'est sur le graphe d'objets
- Quels objets puis-je voir ?
  - Le polygone est encapsulé



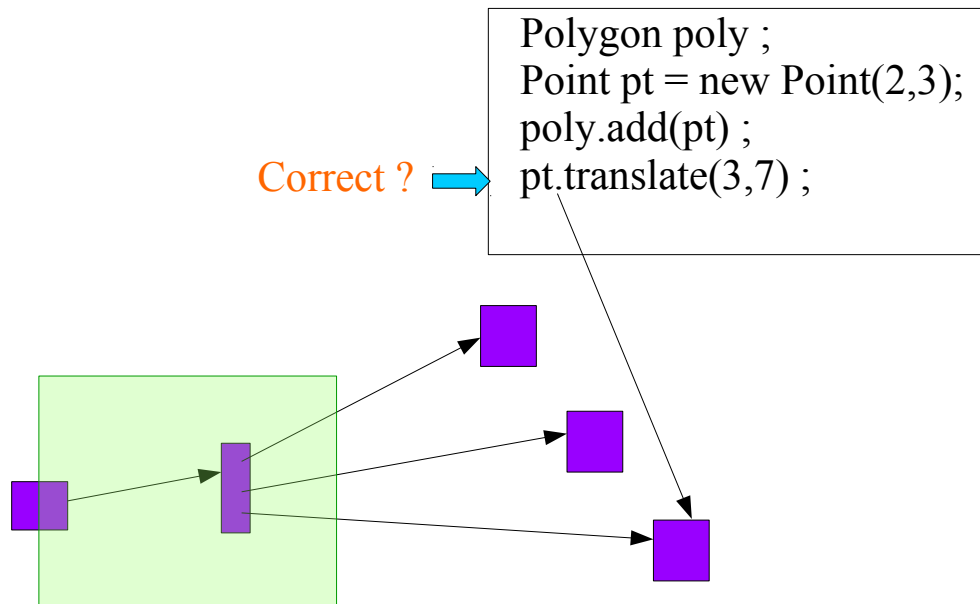
```
public class Point{  
    public int x,y ;  
    public Point(int x, int y ) {  
        this.x = x ;  
        this.y = y ;  
    }  
    public void translate(int dx, int dy ) {  
        this.x += dx ;  
        this.y += dy ;  
    }  
}  
  
public class Polygon{  
    private Point points[];  
    private int npoints;  
  
    public Polygon() {  
        this.points = new Point[4] ;  
    }  
    public void addPoint(Point pt ) {  
        this.points[npoints++] = pt ;  
    }  
}
```

- Encapsulation
  - C'est sur le graphe d'objets
- Quels objets puis-je voir ?
  - Le polygone est encapsulé
  - **Vous pouvez changer l'implémentation**
  - **Sans rien changer d'autre !**

```
public class Point {  
    public int x,y ;  
    public Point(int x, int y ) {  
        this.x = x ;  
        this.y = y ;  
    }  
    public void translate(int dx, int dy ) {  
        this.x += dx ;  
        this.y += dy ;  
    }  
}  
  
public class Polygon {  
    private LinkedList points;  
  
    public Polygon(Point[] pts ) {  
        this.points = new LinkedList() ;  
    }  
    public void addPoint(Point pt ) {  
        this.points.append(pt) ;  
    }  
}
```

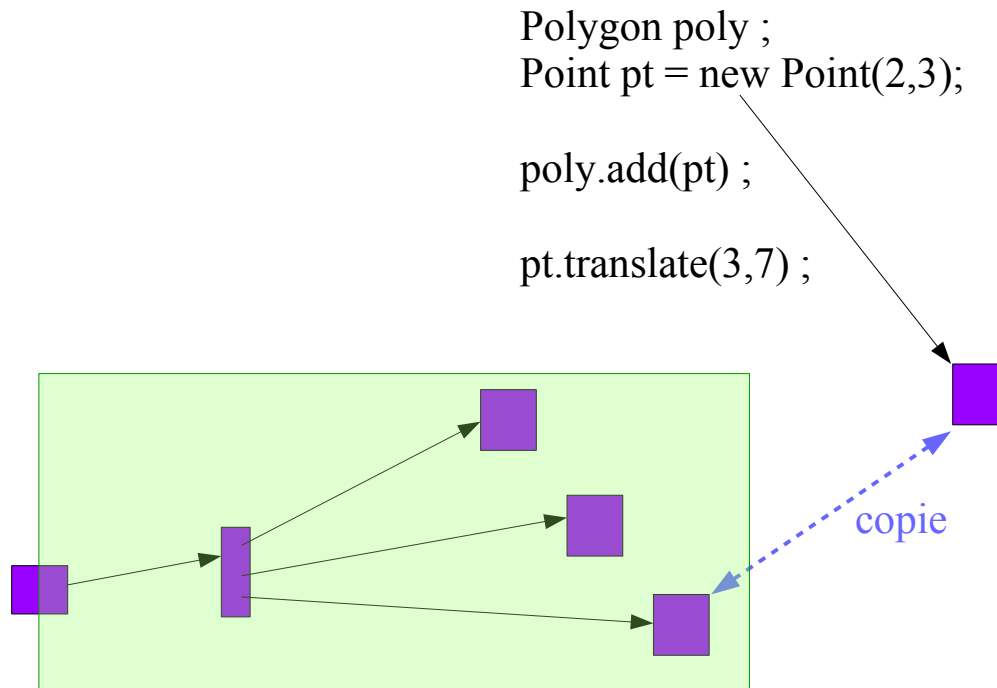


- Encapsulation
  - C'est sur le graphe d'objets
- Quels objets puis-je voir ?
  - Le polygone est encapsulé
  - **Attention** : aliasing possible sur les points



```
public class Point{  
    public int x,y ;  
    public Point(int x, int y ) {  
        this.x = x ;  
        this.y = y ;  
    }  
    public void translate(int dx, int dy ) {  
        this.x += dx ;  
        this.y += dy ;  
    }  
}  
  
public class Polygon{  
    private Point points[];  
    private int npoints;  
  
    public Polygon(Point[] pts ) {  
        this.points = new Point[4] ;  
    }  
    public void addPoint(Point pt ) {  
        points[npoints++] = pt ;  
    }  
}
```

- Encapsulation
  - C'est sur le graphe d'objets
- Quels objets puis-je voir ?
  - Le polygone est totalement encapsulé
  - Par copie des points

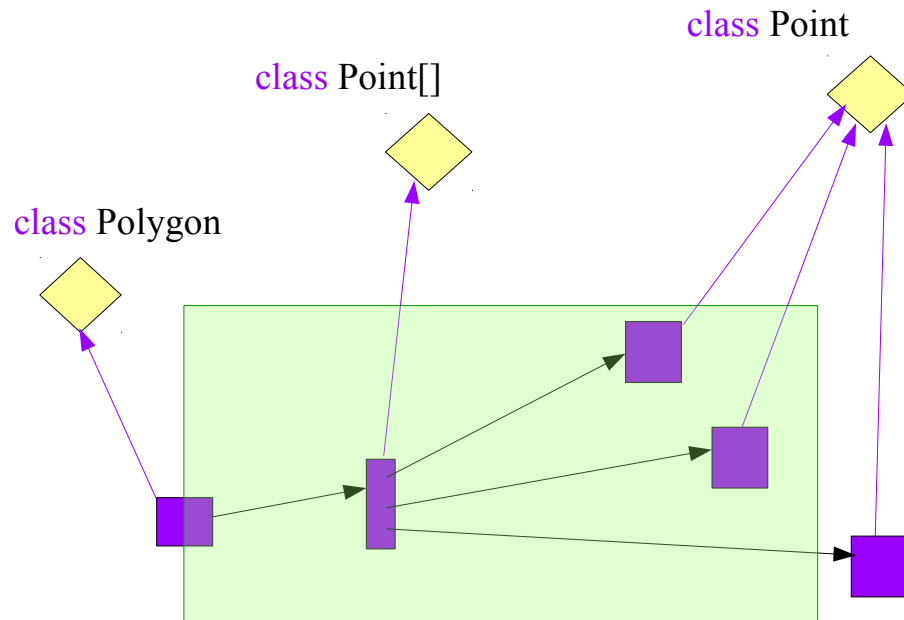


```
public class Point{  
    public int x,y ;  
    public Point(int x, int y ) {  
        this.x = x ;  
        this.y = y ;  
    }  
    public void translate(int dx, int dy ) {  
        this.x += dx ;  
        this.y += dy ;  
    }  
}  
  
public class Polygon{  
    private Point points[];  
    private int npoints;  
  
    public Polygon(Point[] pts ) {  
        this.points = new Point[4] ;  
    }  
    public void addPoint(Point pt ) {  
        points[npoints++] = new Point(pt) ;  
    }  
}
```

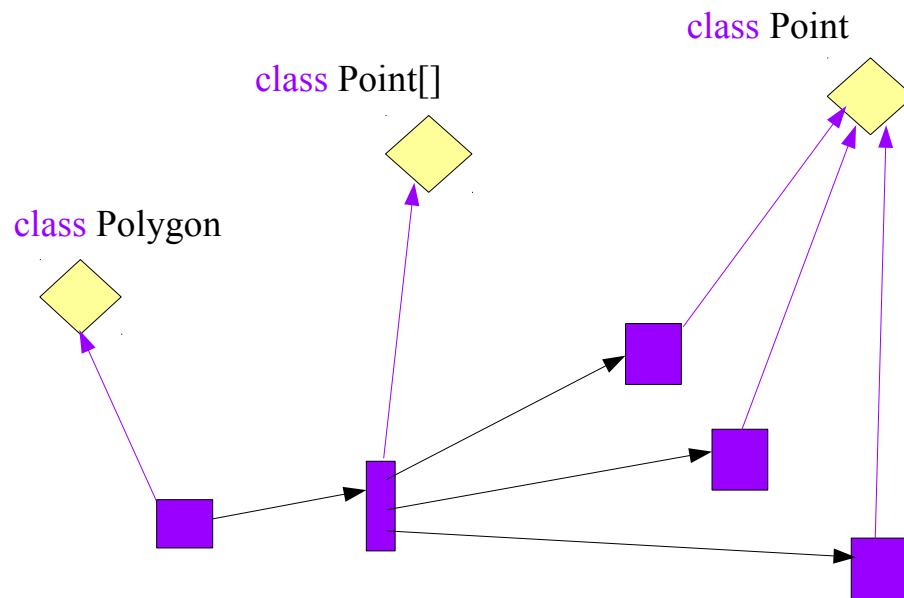
- Encapsulation
  - C'est sur le graphe d'objets
- Qu'est-ce qu'un polygone ?
  - Cela dépend de ce que vous voulez !
  - Vous pouvez donner le choix : copie ou pas

```
public class Point {  
    private int x,y ;  
    public Point(int x, int y ) {  
        this.x = x ;  
        this.y = y ;  
    }  
    public void translate(int dx, int dy ) {  
        this.x += dx ;  
        this.y += dy ;  
    }  
}
```

```
public class Polygon {  
    private Point points[];  
    private int npoints;  
  
    public Polygon(Point[] pts ) {  
        this.points = new Point[4] ;  
    }  
    public void addPoint(Point pt ) {  
        points[npoints++] = pt ;  
    }  
    public void copyPoint(Point pt ) {  
        points[npoints++] = new Point(pt) ;  
    }  
}
```



- C'est quoi une classe à l'exécution ?
  - Elle est créée par la Java Plateforme...
  - En lisant le fichier « class » correspondant (classfile)
  - Mais est-ce un objet ?



```
class Point{
    int x,y ;
    Point(int x, int y ) {
        this.x = x ;
        this.y = y ;
    }
    void translate(int dx, int dy ) {
        this.x += dx ;
        this.y += dy ;
    }
}
Point p = new Point();

Class c = p.getClass();
```



# Le Paradigme Objet

41

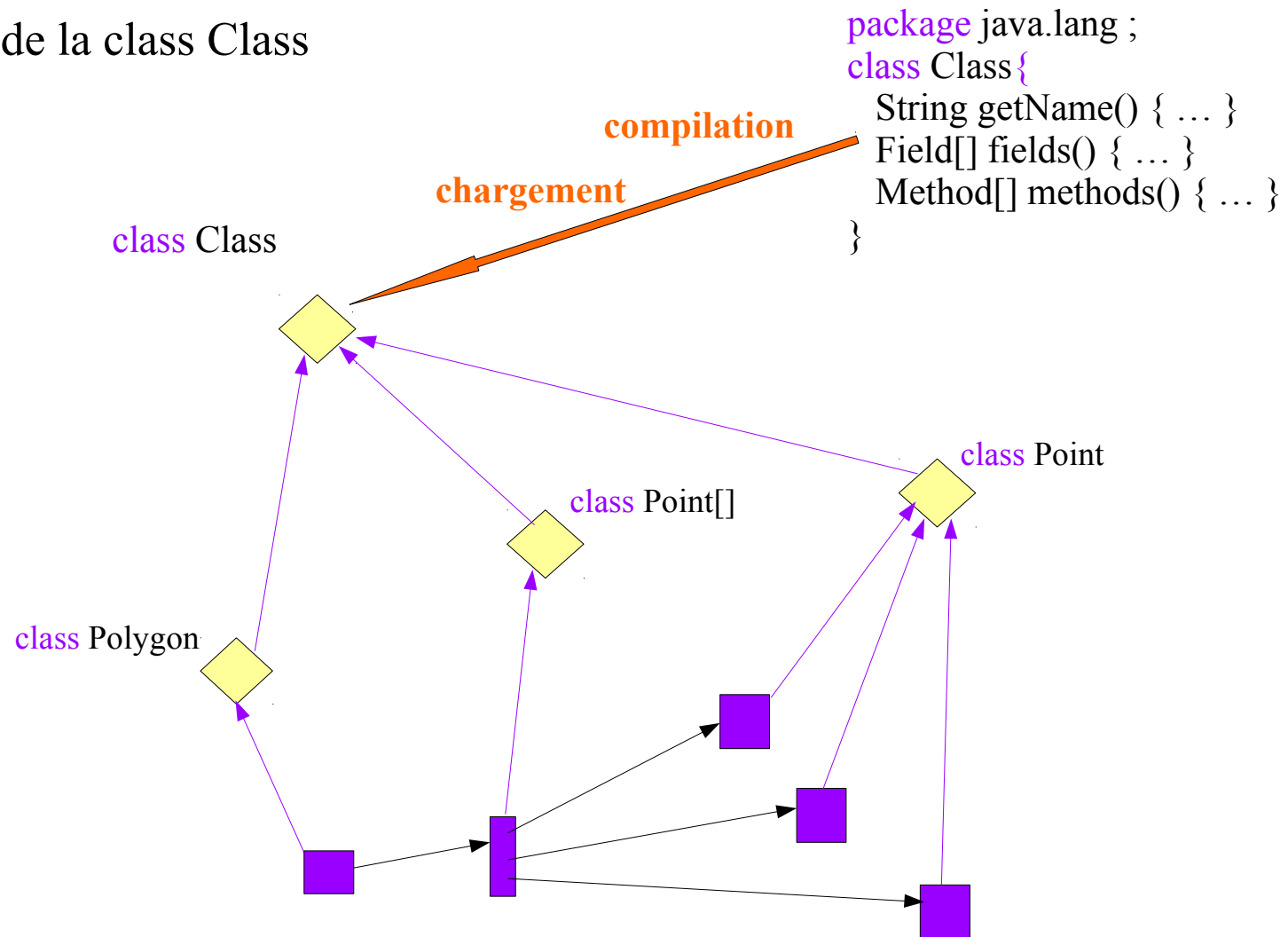
- Tout est objet
  - Les classes sont des objets
  - Elles sont instances de la class Class

Point p = new Point();

Class c = p.getClass();

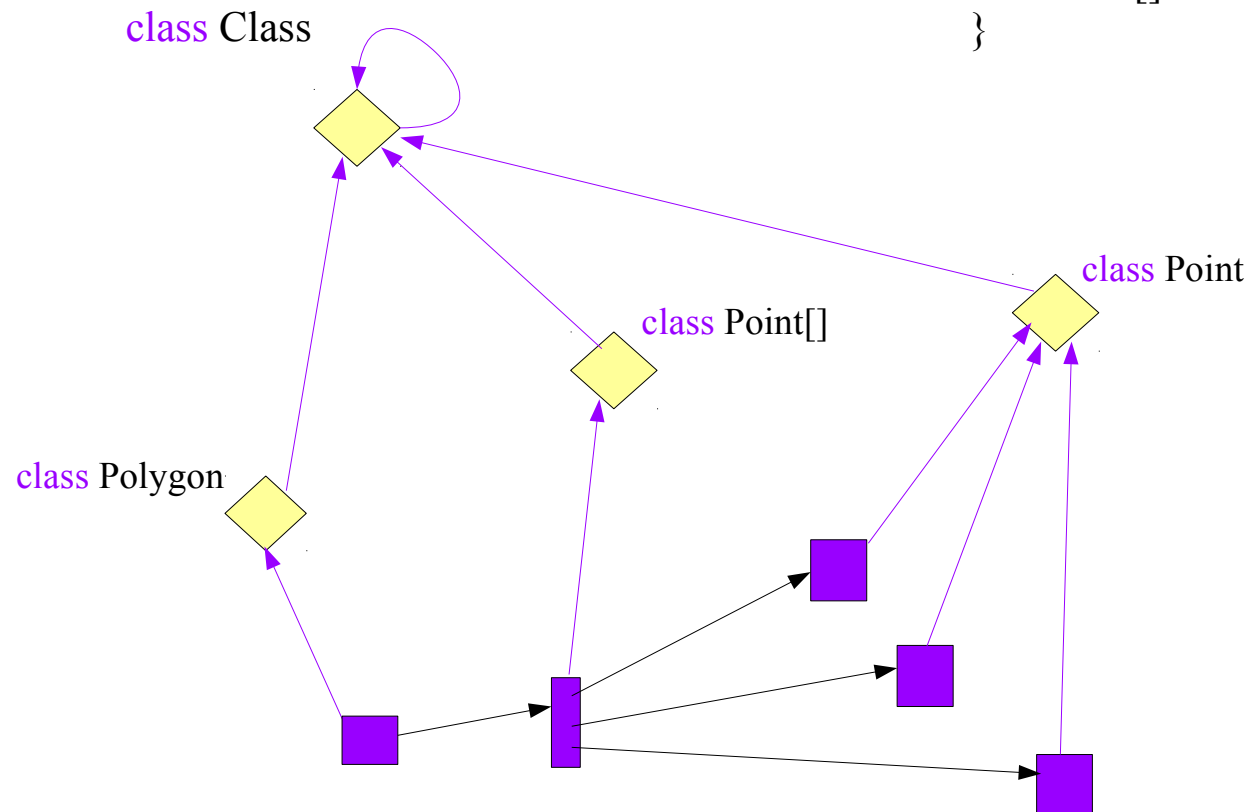
Class cc = c.getClass();

assert(cc == cc.getClass());



- Tout est objet
  - La class Class décrit donc ce qu'est une classe
  - Elle se décrit donc elle-même

```
package java.lang ;  
class Class {  
    String getName() { ... }  
    Field[] fields() { ... }  
    Method[] methods() { ... }  
}
```



- Tout est objet
  - La class Class décrit donc ce qu'est une classe
  - Elle se décrit donc elle-même
- Essayer, c'est l'adopter
  - Créez une classe avec des champs et des methodes
  - Ecrivez une methode main qui imprime le source
    - Le package, le nom de la classe
    - Les champs
    - Les constructeurs et les methodes

```
package ricm.oop ;  
class Point{  
    int x ;  
    int y ;  
    ...  
    public static void main(String[] arg) {  
        Point p = new Point() ;  
        Class cls = p.getClass() ;  
  
        System.out.print("class ") ;  
        System.out.println(cls.getName()) ;  
        Field[] fields = cls.fields() ;  
        ....  
    }  
}
```

- Le cycle de vie des objets
  - Les objets naissent, vivent, et sombrent dans l'oubli

```
void main(String[] args) {  
    Point p = new Point();  
    p.translate(2,4);  
}
```

Naissance d'un objet...

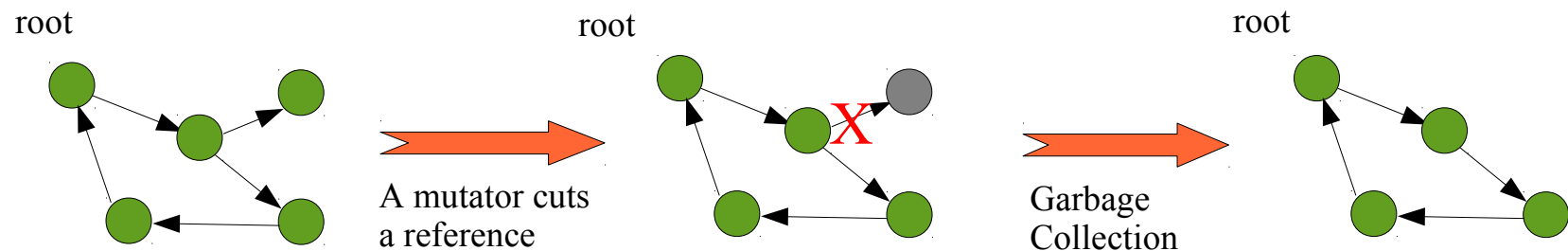
```
→ p = new Point();  
   p.translate(5,-8);  
   return 0;  
}
```

Naissance d'un autre objet...

Mais aussi oubli d'un objet...

Que devient-il ?

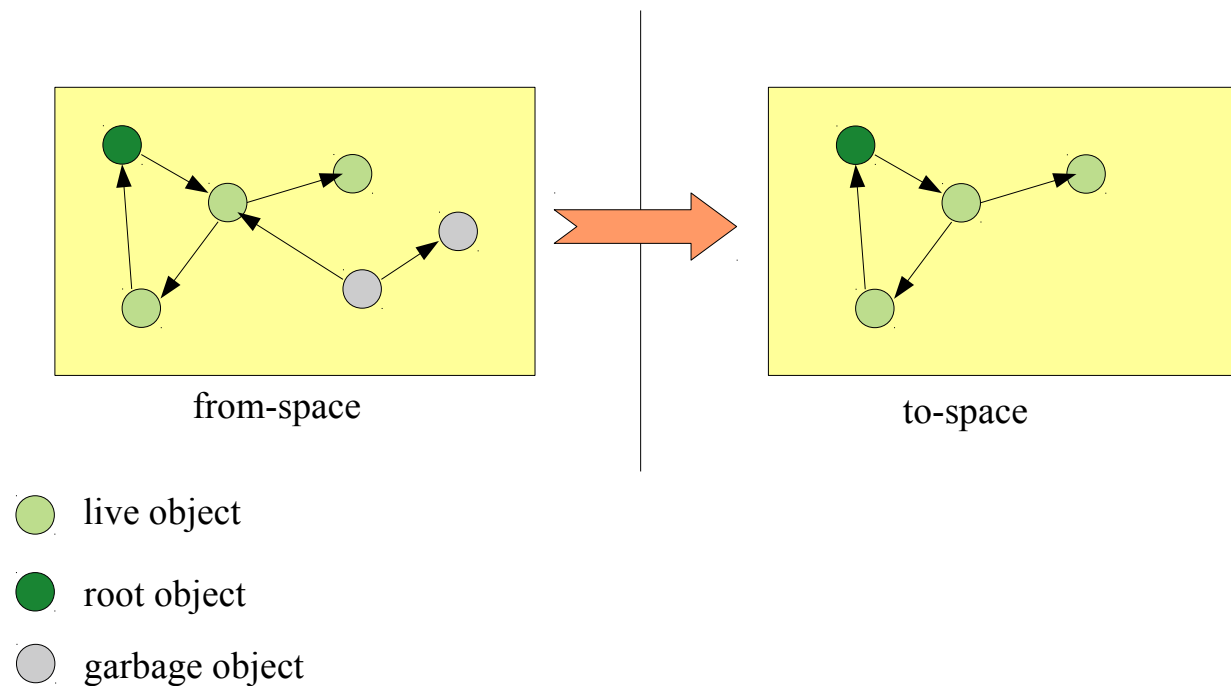
- Le cycle de vie des objets
  - Les objets naissent, vivent, et sombrent dans l'oubli
  - L'oubli entraîne la mort de l'objet
- Ramasse-Miettes (Garbage Collector)
  - Collecte les objets morts...
  - Un objet est vivant si il est atteignable depuis une racine de persistance
  - Les racines sont la pile d'appels et les variables statiques



- Processus automatique et « *durable* »
  - Détecte les objets oubliés et les recycle
- Défis de conception
  - Performance
    - Limiter le surcoût, donc ne pas ramassez les miettes trop souvent
    - Limiter la taille mémoire, donc ramassez les miettes souvent
  - Correction
    - Ne jamais détruire un objet vivant
  - Implémentation en générale difficile
    - Une conception relativement simple – Scavenger Design

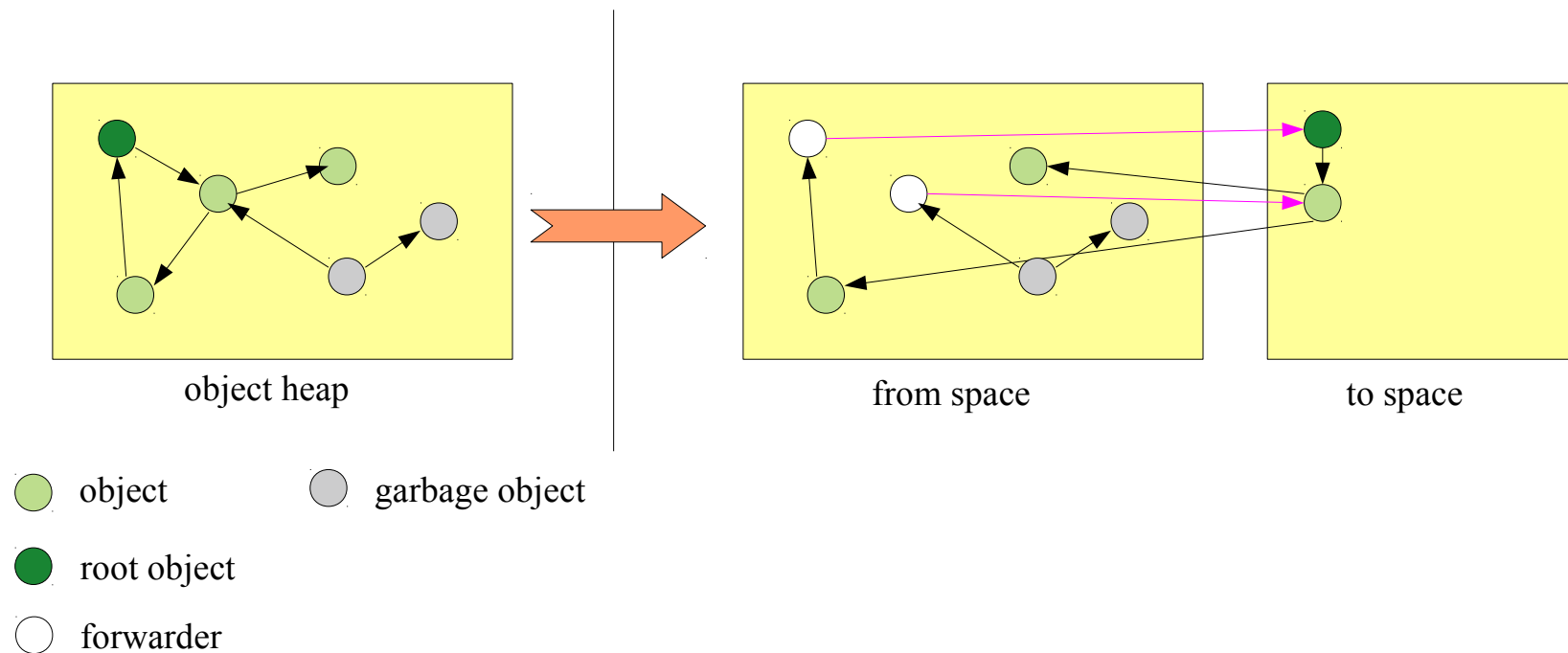
- Scavenger – les principes

- Utilise deux tas d'objet, appelés *from-space* et *to-space*
- Copie les objets vivant du from-space vers le to-space
- Inverse les rôles des deux tas



- Scavenger – quelques détails

- Laisser un *forwarder* en place lorsque l'on copie un objet
  - Permet de maintenir le partage d'objet dans le graphe
  - Permet de gérer les cycles
- Utiliser le *to-space* comme pile de récursion





- Prenez en charge votre apprentissage
  - Vous devez comprendre en details tout le contenu de cette présentation
  - Vous devez mettre en pratique
    - Ecrivez du code qui illustre tous les concepts et mécanismes
    - Suivez l'exécution sous debugger
  - Complétez les transparents avec vos propres notes
  - Posez des questions pour mieux comprendre