

**R1.01**

## **INITIATION AU DÉVELOPPEMENT**

---

### **Cours 4, partie 2 : Agrégation et composition**

*(résumé dans Mémento x)*

**Hervé Blanchon & Anne Lejeune**

Université Grenoble Alpes

IUT 2 – Département Informatique

# Sommaire

---

 Attribut de classe de type classe

 autour des classes Point et Rectangle

 Classe Rectangle avec Points en **composition**

 **RectangleCompose**

 Classe Rectangle avec Points en **agrégation**

 **RectangleAgregge**

 Agrégation et composition

 ce qu'il faut retenir

Exemple autour de classes Point et Rectangle

**UN ATTRIBUT D'UNE CLASSE EST DE  
TYPE CLASSE**

 On dispose d'une classe Point définie comme suit :

```
public class Point {
    // attributs
    private int x;
    private int y;

    // constructeurs
    public Point() {
        x = 0;
        y = 0;
    }
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    // getters et setters des attributs
    public int getX() {return x;}
    public void setX(int x) {
        this.x = x;
    }
    public int getY() {return y;}
    public void setY(int y) {
        this.y = y;
    }
}
```

```
/**
 * Déplacement des coordonnées
 *      en abscisse et en ordonnée
 * @param dx déplacement en abscisse
 * @param dy déplacement en ordonnée
 */
public void deplace(int dx, int dy) {
    x = x + dx;
    y = y + dy;
}

/**
 * Affichage avec printXX() en mode mathématique
 * @return (x, y)
 */
@Override
public String toString() {
    return "(" + x + ", " + y + ")";
}
}
```

# Classe Rectangle

On veut faire une classe Rectangle avec deux attributs

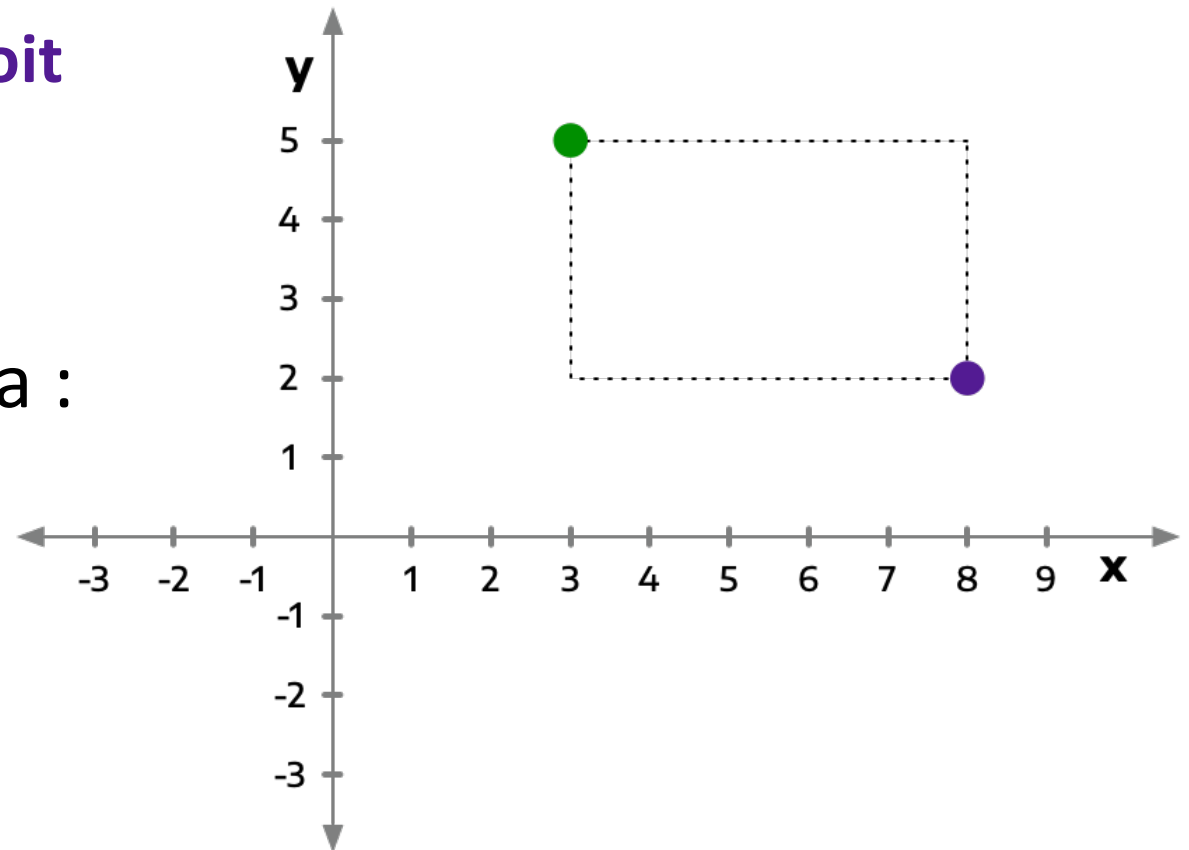
un Point supérieur gauche

un Point inférieur droit

Dans l'exemple, on a :

(3, 5)

(8, 2)



# Ébauche de la classe Rectangle

```
public class Rectangle {

    private Point pointHG; // Point haut-gauche
    private Point pointBD; // Point bas-droit

    public Rectangle(Point pointHG,
                    Point pointBD) {
        // on a le choix
        // option 1 : les pointHG et pointBD n'existent
        //                que si ce Rectangle existe
        // --> COMPOSITION
        // option 2 : les pointHG et pointBD existent
        //                indépendamment de ce Rectangle
        // --> AGRÉGATION
    }

    public Point getPointHG() {
        return pointHG;
    }

    public Point getPointBD() {
        return pointBD;
    }

    public int getLongueurAbscisse() {
        return getPointBD().getX()
               - getPointHG().getX();
    }

    public int getLongueurOrdonnee() {
        return getPointHG().getY()
               - getPointBD().getY();
    }

    public int getSurface() {
        return getLongueurAbscisse()
               * getLongueurOrdonnee();
    }

    public int getPerimetre() {
        return 2 * (getLongueurAbscisse()
                   + getLongueurOrdonnee());
    }

    public void deplacePointHG(int dx, int dy) {
        getPointHG().deplace(dx, dy);
    }

    public void deplacePointBD(int dx, int dy) {
        getPointBD().deplace(dx, dy);
    }
}
```

# CLASSE Rectangle AVEC Point EN COMPOSITION

# Classe Rectangle avec composition

```
public class RectangleCompose {  
  
    private Point pointHG; // Point haut-gauche  
    private Point pointBD; // Point bas-droit  
  
    public RectangleCompose(Point pointHG,  
                             Point pointBD) {  
        this.pointHG = new Point(pointHG.getX(),  
                                   pointHG.getY());  
        this.pointBD = new Point(pointBD.getX(),  
                                   pointBD.getY());  
        // Les attributs pointHG et pointBD pointent  
        // sur des points construits exclusivement  
        // pour ce RectangleCompose (new Point())  
        // Ce RectangleCompose est modifiable  
        // uniquement par des méthodes de la classe  
    }  
  
    public Point getPointHG() {  
        return pointHG;  
    }  
  
    public Point getPointBD() {  
        return pointBD;  
    }  
}
```

```
    public int getLongueurAbscisse() {  
        return getPointBD().getX()  
               - getPointHG().getX();  
    }  
  
    public int getLongueurOrdonnee() {  
        return getPointHG().getY()  
               - getPointBD().getY();  
    }  
  
    public int getSurface() {  
        return getLongueurAbscisse()  
               * getLongueurOrdonnee();  
    }  
  
    public int getPerimetre() {  
        return 2 * (getLongueurAbscisse()  
                    + getLongueurOrdonnee());  
    }  
  
    public void deplacePointHG(int dx, int dy) {  
        getPointHG().deplace(dx, dy);  
    }  
  
    public void deplacePointBD(int dx, int dy) {  
        getPointBD().deplace(dx, dy);  
    }  
}
```



# Ce qui se passe avec RectangleCompose

```
1 public class RectangleCompose_Main {
2
3     public static void main(String[] args) {
4         Point ptHG = new Point(8, 8);
5         Point ptBD = new Point(12, 4);
6
7         System.out.println("ptHG : " + ptHG + ", ptBD : " + ptBD);
8         System.out.println("rectangle composé rect1 avec les coordonnées de ptHG et ptBD");
9         RectangleCompose rect1 = new RectangleCompose(ptHG, ptBD);
10        RectangleUtilitaire.afficherRectangle(rect1);
11
12        System.out.println("déplacement de ptHG");
13        ptHG.deplace(-4, -3);
14        System.out.println("ptHG : " + ptHG + ", ptBD : " + ptBD);
15        RectangleUtilitaire.afficherRectangle(rect1);
16
17        System.out.println("déplacement du pointHG de rect1");
18        rect1.deplacePointHG(-4, -3);
19        System.out.println("ptHG : " + ptHG + ", ptBD : " + ptBD);
20        RectangleUtilitaire.afficherRectangle(rect1);
21    }
22 }
```

*procédure définie dans une classe  
RectangleUtilitaire (cf. ANNEXE)*



## trace

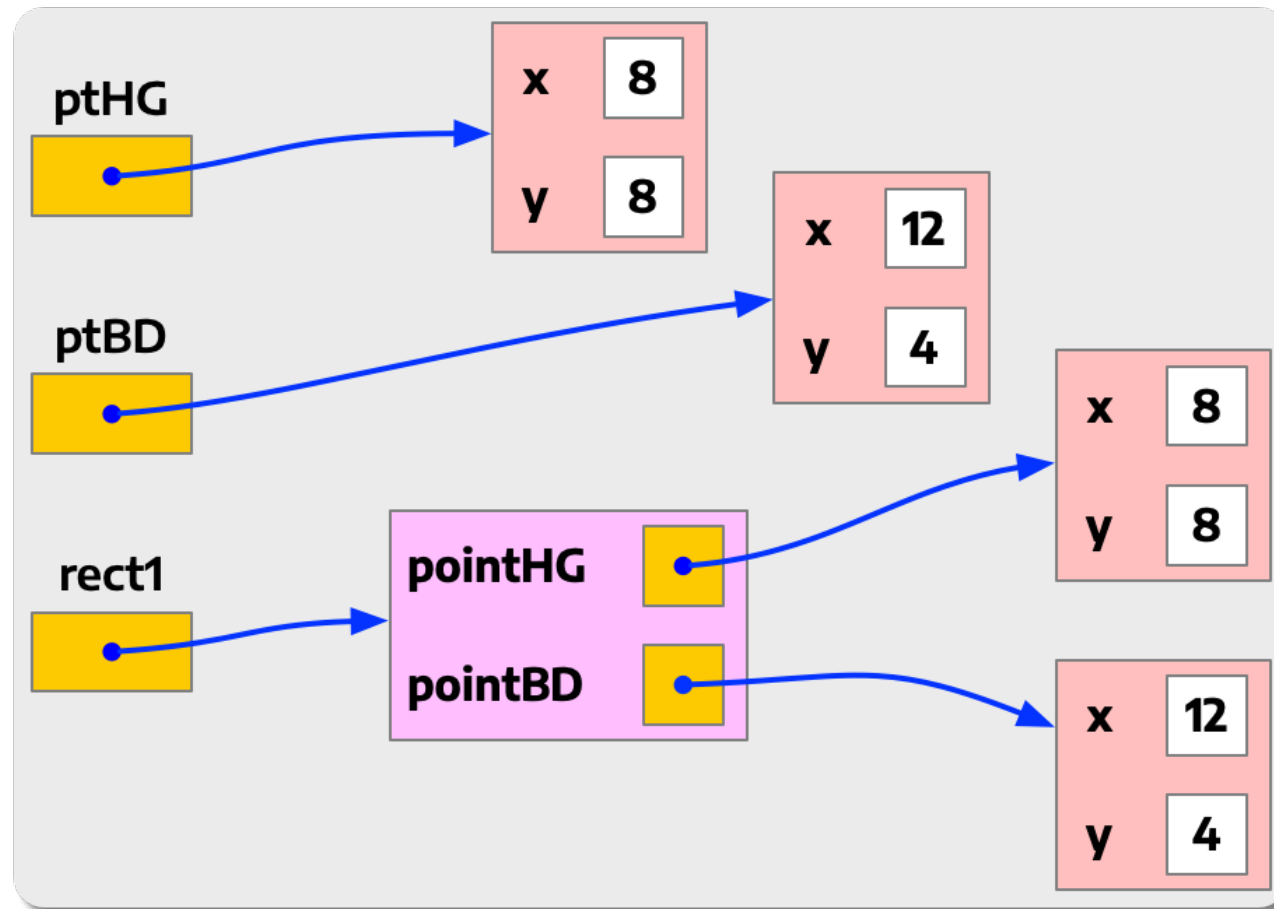
```
7 ptHG : (8, 8), ptBD : (12, 4)
8 rectangle composé rect1 avec les coordonnées de ptHG et ptBD
10 Rectangle composé :
   - Point haut gauche : (8, 8) // pointHG exclusif à rect1 (coordonnées de ptHG)
   - Point bas droit : (12, 4) // pointBD exclusif à rect1 (coordonnées de ptBD)
   - Surface : 16
   - Périmètre : 16

12 déplacement de ptHG
14 ptHG : (4, 5), ptBD : (12, 4) // coordonnées de ptHG modifiées
15 Rectangle composé :
   - Point haut gauche : (8, 8) // coordonnées de pointHG de rect1 non modifiées
   - Point bas droit : (12, 4)
   - Surface : 16
   - Périmètre : 16

17 déplacement du pointHG de rect1
19 ptHG : (4, 5), ptBD : (12, 4)
20 Rectangle composé :
   - Point haut gauche : (4, 5) // coordonnées de pointHG de rect1 modifiées
   - Point bas droit : (12, 4)
   - Surface : 8
   - Périmètre : 18
```

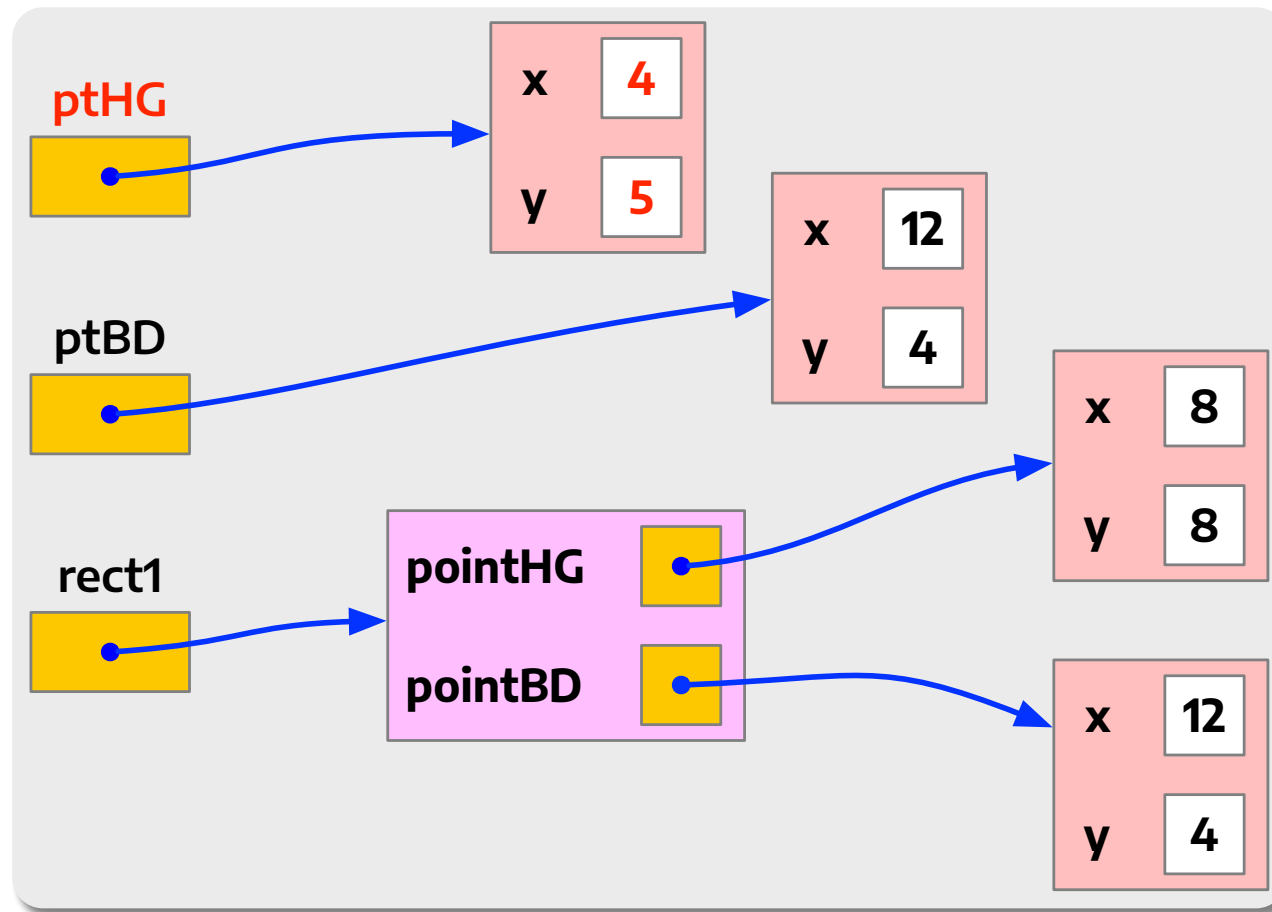
# État de la mémoire (RectangleCompose)

```
4 Point ptHG = new Point(8, 8);  
5 Point ptBD = new Point(12, 4);  
9 RectangleCompose rect1 = new RectangleCompose(ptHG, ptBD);
```



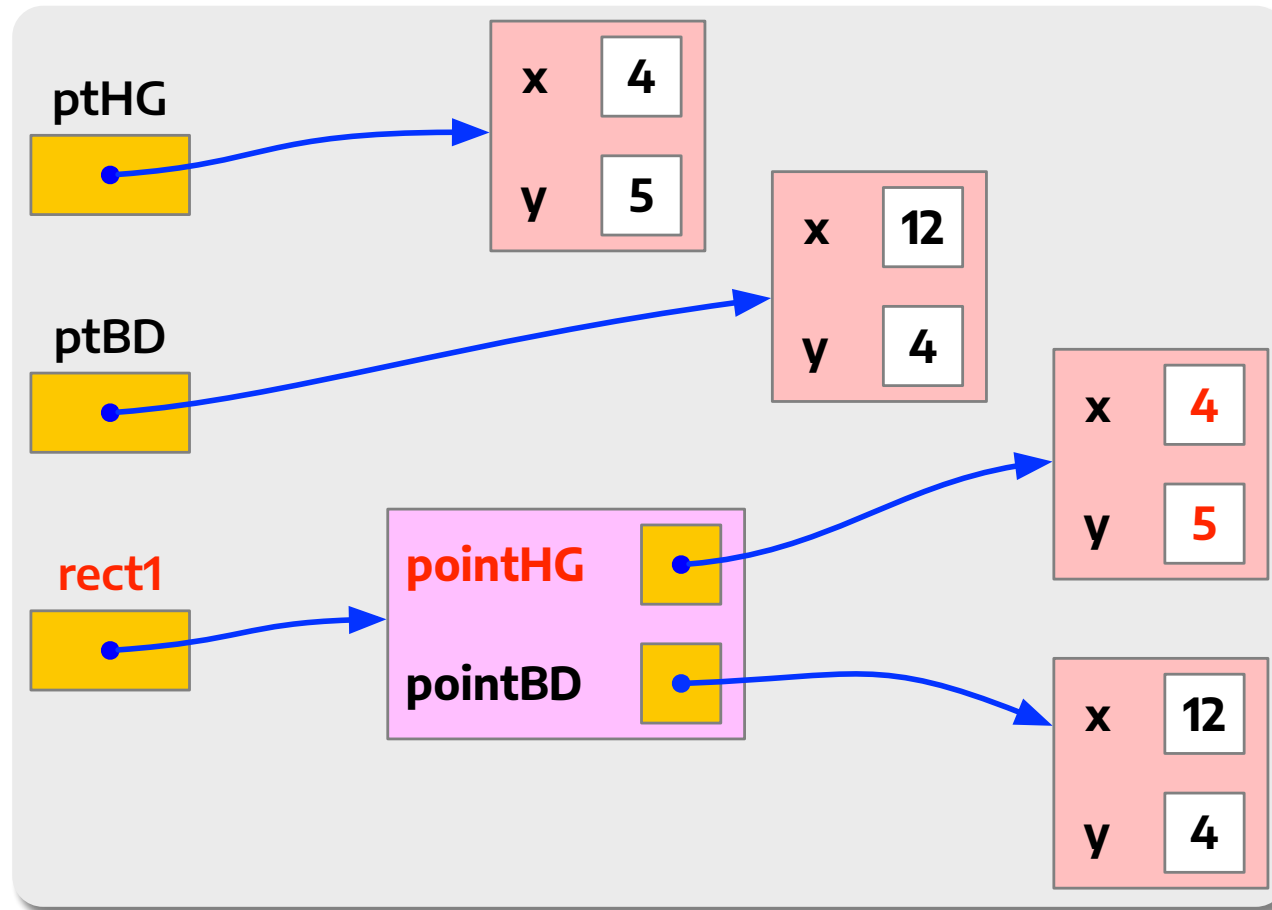
# État de la mémoire (RectangleCompose)

13 ptHG.deplace(-4, -3);



# État de la mémoire (RectangleCompose)

18 `rect1.deplacePointHG(-4, -3);`



# CLASSE Rectangle AVEC Point EN AGRÉGATION

# Classe Rectangle avec agrégation

```
public class RectangleAgrege {  
  
    private Point pointHG; // Point haut-gauche  
    private Point pointBD; // Point bas-droit  
  
    public RectangleAgrege(Point pointHG,  
                           Point pointBD) {  
        this.pointHG = pointHG;  
        this.pointBD = pointBD;  
        // Les attributs pointHG et pointBD pointent  
        // sur des points qui existent indépendamment  
        // de ce RectangleAgrege  
        // Toute modification des points pointés  
        // modifie donc ce RectangleAgrege  
    }  
  
    public Point getPointHG() {  
        return pointHG;  
    }  
  
    public Point getPointBD() {  
        return pointBD;  
    }  
}
```

```
    public int getLongueurAbscisse() {  
        return getPointBD().getX()  
            - getPointHG().getX();  
    }  
  
    public int getLongueurOrdonnee() {  
        return getPointHG().getY()  
            - getPointBD().getY();  
    }  
  
    public int getSurface() {  
        return getLongueurAbscisse()  
            * getLongueurOrdonnee();  
    }  
  
    public int getPerimetre() {  
        return 2 * (getLongueurAbscisse()  
            + getLongueurOrdonnee());  
    }  
  
    public void deplacePointHG(int dx, int dy) {  
        getPointHG().deplace(dx, dy);  
    }  
  
    public void deplacePointBD(int dx, int dy) {  
        getPointBD().deplace(dx, dy);  
    }  
}
```

# Ce qui se passe avec RectangleAgrege

```
1 public class RectangleAgrege_Main {
2
3     public static void main(String[] args) {
4         Point ptHG = new Point(8, 8);
5         Point ptBD = new Point(12,4);
6
7         System.out.println("ptHG : " + ptHG + ", ptBD : " + ptBD);
8         System.out.println("rectangle agrégé rect1 avec ptHG et ptBD");
9         RectangleAgrege rect1 = new RectangleAgrege(ptHG, ptBD);
10        RectangleUtilitaire.afficherRectangle(rect1);
11
12        System.out.println("déplacement de ptHG");
13        ptHG.deplace(-4, -3);
14        System.out.println("ptHG : " + ptHG + ", ptBD : " + ptBD);
15        RectangleUtilitaire.afficherRectangle(rect1);
16
17        System.out.println("déplacement du pointHG de rect1");
18        rect1.deplacePointHG(4, 3);
19        System.out.println("ptHG : " + ptHG + ", ptBD : " + ptBD);
20        RectangleUtilitaire.afficherRectangle(rect1);
21    }
22 }
```

*procédure définie dans une classe  
RectangleUtilitaire (cf. ANNEXE)*





## trace

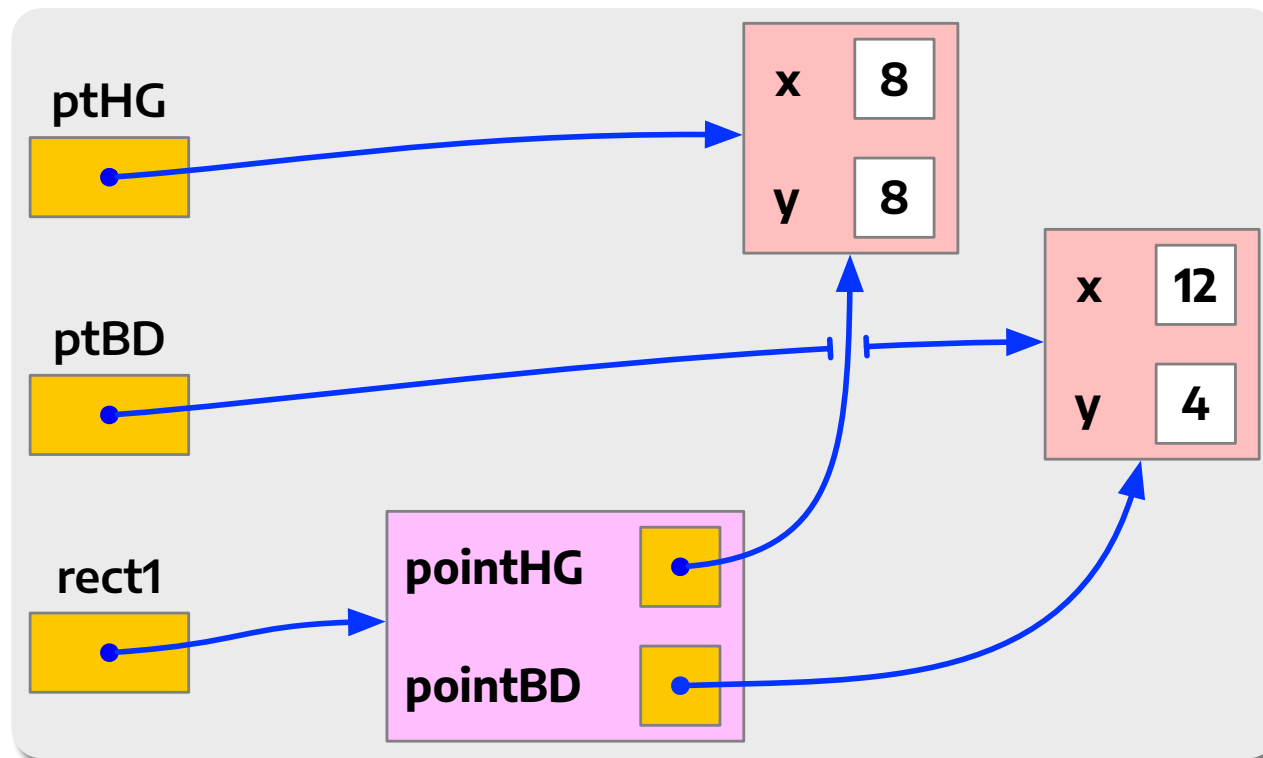
```
7 ptHG : (8, 8), ptBD : (12, 4)
8 rectangle agrégé rect1 avec les coordonnées de ptHG et ptBD
10 Rectangle agrégé :
    - Point haut gauche : (8, 8) // pointHG qui est ptHG
    - Point bas droit : (12, 4) // pointBD qui est ptBD
    - Surface : 16
    - Périmètre : 16

12 déplacement de ptHG
14 ptHG : (4, 5), ptBD : (12, 4) // coordonnées de ptHG modifiées
15 Rectangle agrégé :
    - Point haut gauche : (4, 5) // coordonnées de pointHG de rect1 modifiées (c'est ptHG)
    - Point bas droit : (12, 4)
    - Surface : 8 // attention à corriger
    - Périmètre : 18 // attention à corriger

17 déplacement du pointHG de rect1 // on déplace en fait ptHG
19 ptHG : (8, 8), ptBD : (12, 4)
20 Rectangle agrégé :
    - Point haut gauche : (8, 8) // coordonnées de pointHG de rect1 modifiées (c'est ptHG)
    - Point bas droit : (12, 4)
    - Surface : 16 // attention à corriger
    - Périmètre : 16 // attention à corriger
```

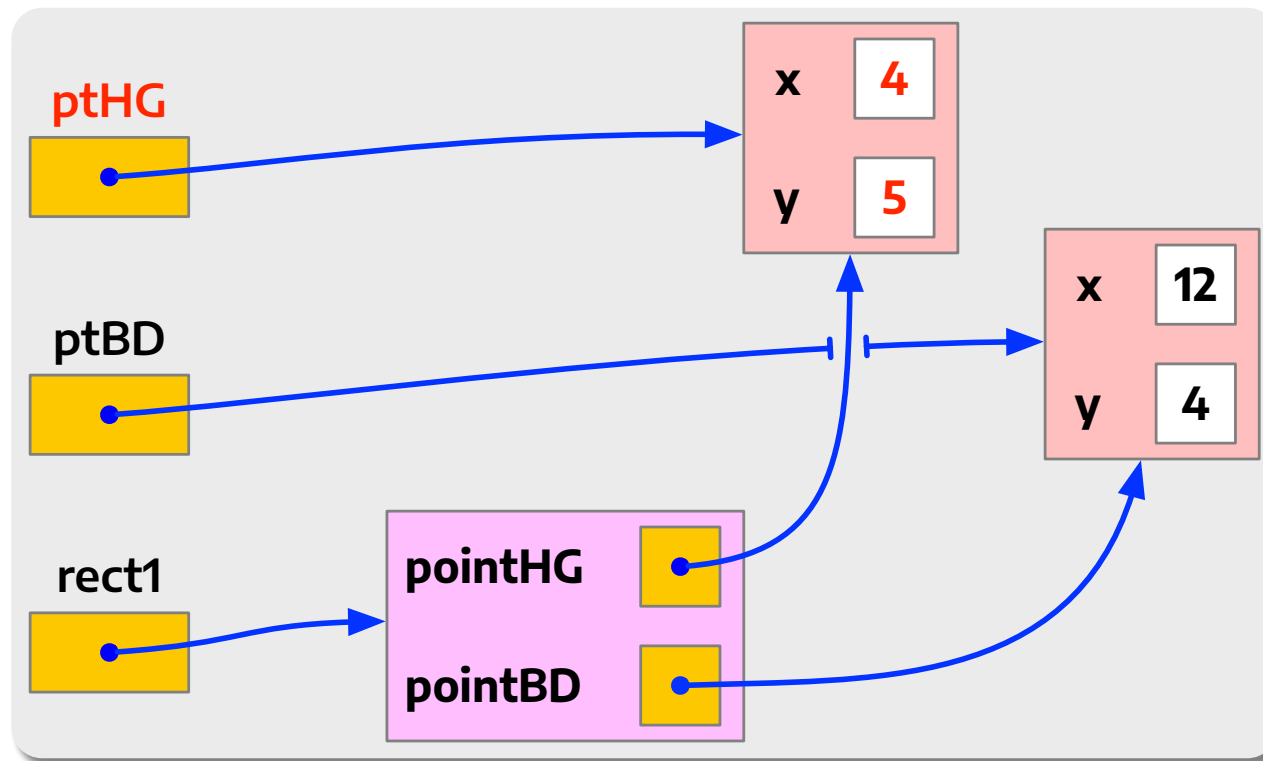
# État de la mémoire (RectangleAgrege)

```
4 Point ptHG = new Point(8, 8);  
5 Point ptBD = new Point(12,4);  
9 RectangleAgrege rect1 = new RectangleAgrege(ptHG, ptBD);
```



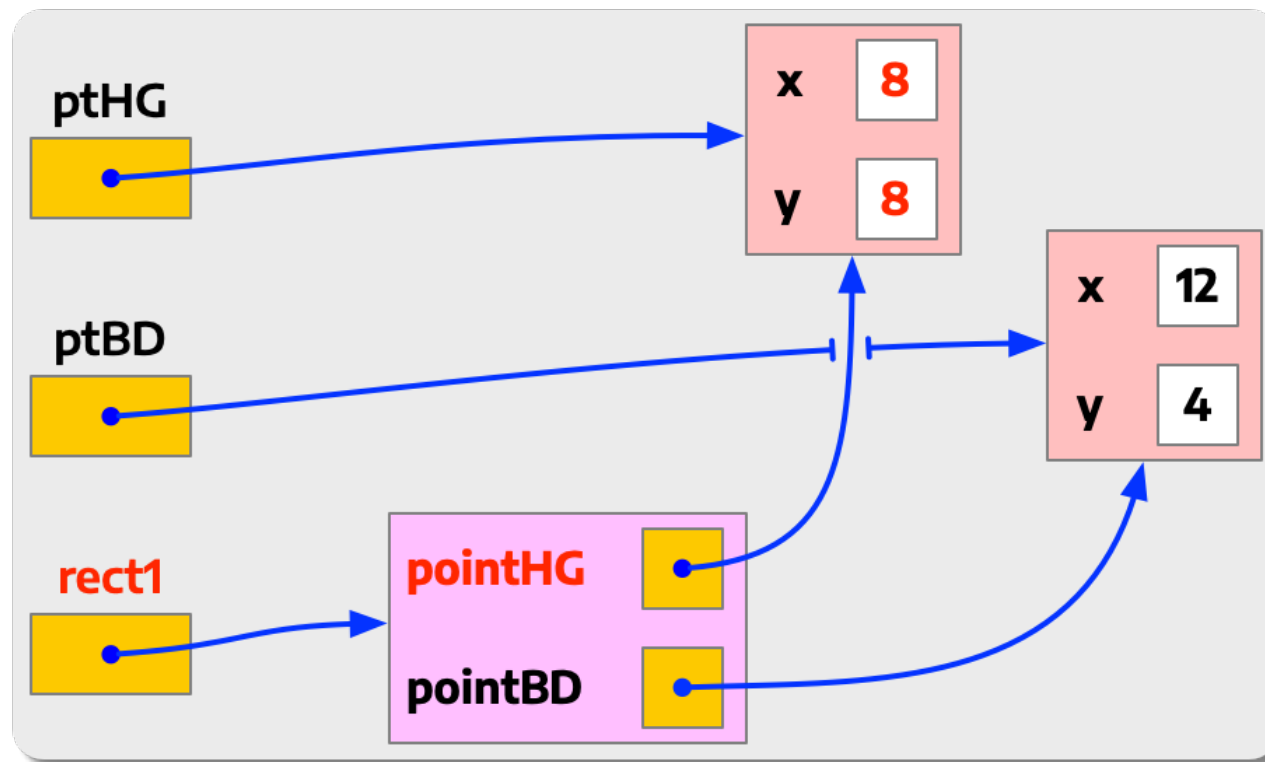
# État de la mémoire (RectangleAgregé)

13 ptHG.deplace(-4, -3);



# État de la mémoire (RectangleAgregé)

18 `rect1.deplacePointHG(4, 3);`







# AGRÉGATION ET COMPOSITION

CE QU'IL FAUT RETENIR

# Vocabulaire

---


## Classe hôte

-  classe qui possède (héberge) un attribut de type classe
-  un attribut de type classe peut être hébergé...
  -  par **agrégation** : dans ce cas l'objet agrégé existe indépendamment de la classe hôte (une seule instance d'objet vue à l'intérieur et à l'extérieur de la classe hôte)
  -  par **composition** (agrégation forte) : dans ce cas l'objet composé n'existe que pour une instance de la classe hôte (une nouvelle instance d'objet propre à la classe hôte)




# Propriétés instanciées

---

## Agrégation

-  implique une relation dans laquelle l'attribut **agrégé** existe indépendamment de l'instance de la classe hôte qui l'héberge
  - ➔ *une instance de l'attribut agrégé peut être partagée par plusieurs instances de la classe hôte*


## Exemple : classe **RectangleAgrege**

-  plusieurs instances de **RectangleAgrege** peuvent partager une même instance **p** d'un **Point**
-  lorsque l'on déplace une instance **p** d'un **Point**, les instances de **RectangleAgrege** qui partagent **p** sont déplacés
-  lorsque l'on déplace un **pointHB** ou **pointBD** d'une instance de **RectangleAgrege**, on déplace une instance **p** de **Point** et donc on déplace aussi toutes les instances de **RectangleAgrege** qui partagent **p**

# Propriétés instanciées




---

## Composition (agrégation forte)

-  implique une relation dans laquelle l'attribut **composé** n'existe que pour l'instance de la classe hôte qui l'héberge

➡ *une instance de l'attribut est propre (exclusive) à une instance de la classe hôte*

## Exemple : classe **RectangleCompose**


-  chaque instance de **RectangleCompose** à l'exclusivité des attributs **pointHB** et **pointBD**
-  lorsque l'on déplace une instance **p** d'un **Point**, il n'y a jamais d'effet sur les instances de **RectangleCompose**
-  lorsque l'on déplace le **pointHB** ou le **pointBD** d'une instance de **RectangleCompose**, on déplace uniquement l'instance concernée, il n'y a pas d'effet sur les autres instances de **RectangleCompose** ni sur les instances de **Point**



# Composition vs. Agrégation - Comment choisir ?

---

 On ne peut pas toujours choisir...

 Un attribut de classe de type `String` ou classe enveloppe (*`Boolean`, `Character`, `Byte`, `Short`, `Integer`, `Long` ou `Double`*) doit toujours être hébergé par composition (pour cause d'immuabilité\*)

 Les cas où le choix est possible seront étudiés en R2.01

\* Rappel : une classe immuable est une classe dont on ne peut pas modifier les attributs

# Une classe avec des attributs immuables

---

```
public class ClasseTestImmuables {  
  
    private String monString;  
    private Integer monInteger;  
  
    public ClasseTestImmuables(String monString, Integer monInteger) {  
        this.monString = monString;  
        this.monInteger = monInteger;  
    }  
  
    public String getMonString() {  
        return monString;  
    }  
    public void setMonString(String monString) {  
        this.monString = monString;  
    }  
    public Integer getMonInteger() {  
        return monInteger;  
    }  
    public void setMonInteger(Integer monInteger) {  
        this.monInteger = monInteger;  
    }  
}
```

# Ce qui se passe avec ClasseTestImmuables

```
1 public class ClasseTestImmuables_Main {
2
3     public static void main(String[] args) {
4         String unString = "variable chaîne initiale";
5         Integer unInteger = 12;
6
7         System.out.println("valeurs initiales des variables locales :");
8         System.out.println(" -> unString : \"" + unString + "\", unInteger : " + unInteger);
9         System.out.println("instanceTemoin construite avec unString et unInteger : ");
10        ClasseTestImmuables instanceTemoin = new ClasseTestImmuables(unString, unInteger);
11        ClasseTestImmuablesUtilitaire.afficherInstance(instanceTemoin);
12        System.out.println("modification des valeurs des variable locales");
13        unString = "variable chaîne modifiée";
14        unInteger = 24;
15        System.out.println("valeurs modifiées des variables locales :");
16        System.out.println(" -> unString : \"" + unString + "\", unInteger : " + unInteger);
17        System.out.println("instanceTemoin après modification des variables locales : ");
18        ClasseTestImmuablesUtilitaire.afficherInstance(instanceTemoin);
19        System.out.println("modification des valeurs des attributs de instanceTemoin avec les setters");
20        instanceTemoin.setMonString("attribut chaîne modifiée");
21        instanceTemoin.setMonInteger(120);
22        System.out.println("valeurs actuelles des variable locales :");
23        System.out.println(" -> unString : \"" + unString + "\", unInteger : " + unInteger);
24        System.out.println("instanceTemoin après modification des valeurs de ses attributs : ");
25        ClasseTestImmuablesUtilitaire.afficherInstance(instanceTemoin);
26    }
27 }
```

procédure définie dans une classe  
*ClasseTestImmuablesUtilitaire* (cf. ANNEXE)



## trace

```
7  valeurs initiales des variables locales :
8  -> unString : "variable chaîne initiale", unInteger : 12

9  instanceTemoin construite avec unString et unInteger :
11 -> attribut monString : "variable chaîne initiale", attribut monInteger : 12

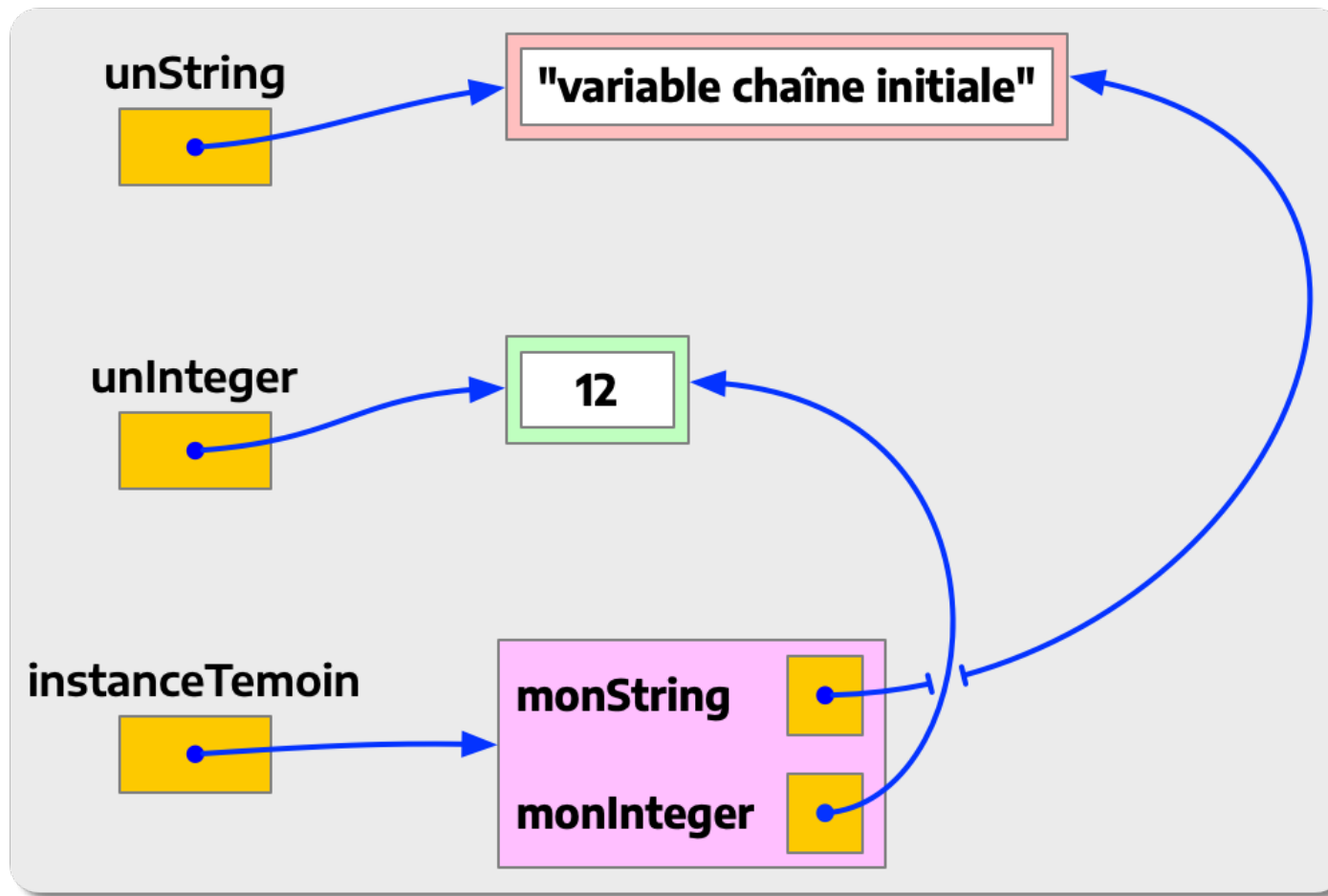
12 modification des valeurs des variable locales
13 valeurs modifiées des variables locales :
15 -> unString : "variable chaîne modifiée", unInteger : 24
16 instanceTemoin après modification des variables locales :
18 -> attribut monString : "variable chaîne initiale", attribut monInteger : 12
    // instanceTemoin non modifiée alors que unString et unInteger le sont

19 modification des valeurs des attributs de instanceTemoin avec les setters
22 valeurs actuelles des variable locales :
23 -> unString : "variable chaîne modifiée", unInteger : 24
    // variables locales non affectées par les modifications sur instanceTemoin

24 instanceTemoin après modification des valeurs de ses attributs :
25 -> attribut monString : "attribut chaîne modifiée", attribut monInteger : 120
    // instanceTemoin bien modifiée par les setter de la classe
```

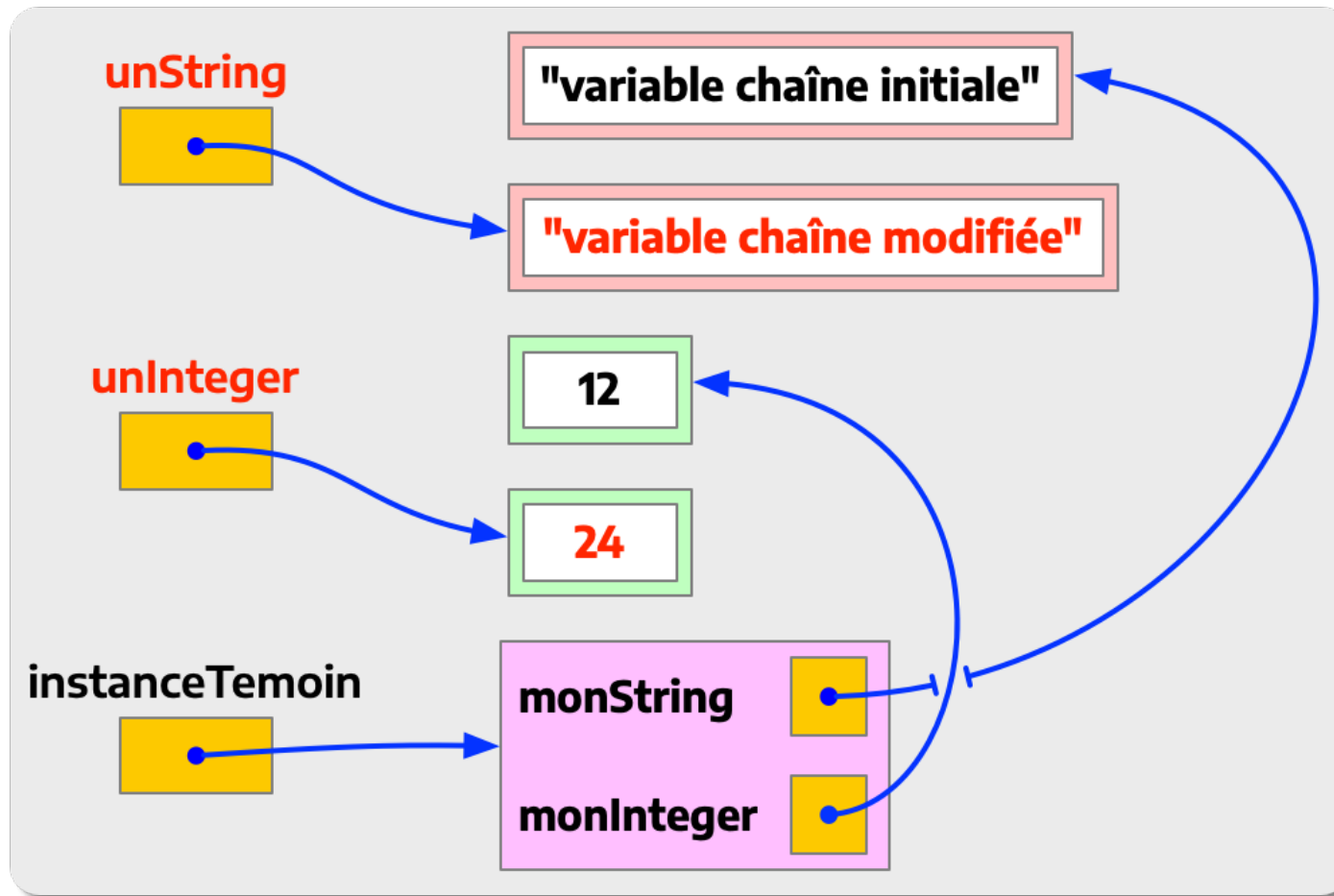
# État de la mémoire ClasseTestImmuables

```
4 String unString = "variable chaîne initiale";  
5 Integer unInteger = 12;  
10 ClasseTestImmuables instanceTemoin = new ClasseTestImmuables(unString, unInteger);
```



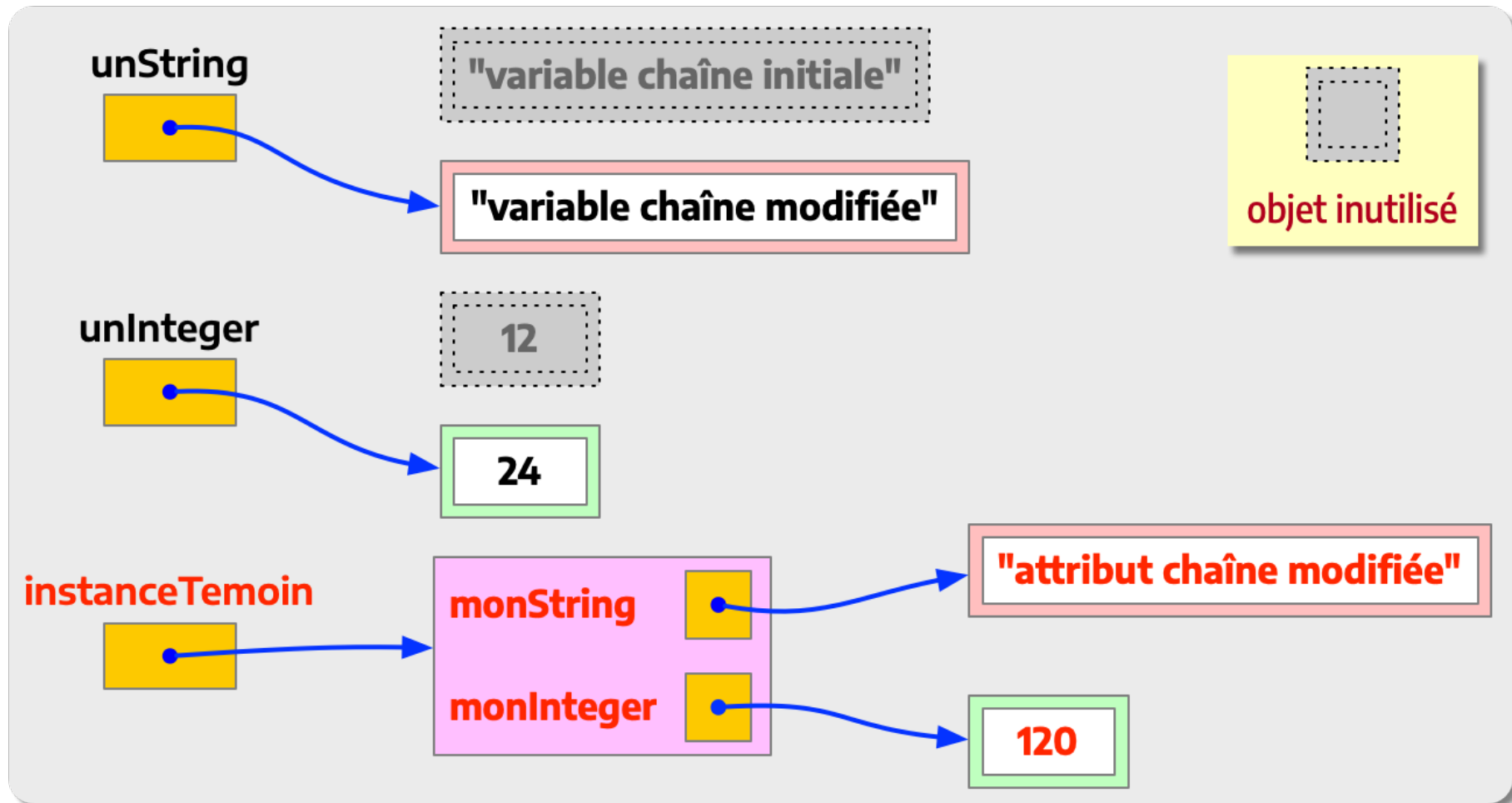
# État de la mémoire ClasseTestImmuables

```
13 unString = "variable chaîne modifiée";  
14 unInteger = 24;
```




# État de la mémoire ClasseTestImmuables

```
20 instanceTemoin.setMonString("attribut chaîne modifiée");  
21 instanceTemoin.setMonInteger(120);
```



# Objet inutilisé ?

---


 Un objet utilisé (utile) est un objet « pointé »

 par une variable

 exemple : objet pointé par unString, ou par instanceTemoin

 par un attribut de classe

 exemple : objet pointé par monString attribut de instanceTemoin


 Un objet inutilisé (inutile) est un objet « non pointé »

 un objet « non pointé » ne peut pas redevenir « pointé »

 il occupe de la place en mémoire pour rien !

 exemple : objet grisés sur la planche précédente

 Objets inutilisés et Java

 il existe dans la machine virtuelle Java, un mécanisme appelé « ramasse miettes » (**garbage collector**) qui libère l'espace mémoire occupé inutilement par des objets « non pointés »



# ANNEXE

# Classe RectangleUtilitaire

---

```
public class RectangleUtilitaire {

    public static void afficherRectangle(RectangleAgrege rectangle) {
        System.out.println("Rectangle agrégé :");
        System.out.println(" - Point haut gauche : " + rectangle.getPointHG());
        System.out.println(" - Point bas droit : " + rectangle.getPointBD());
        System.out.println(" - Surface : " + rectangle.getSurface());
        System.out.println(" - Périmètre : " + rectangle.getPerimetre());
    }

    public static void afficherRectangle(RectangleCompose rectangle) {
        System.out.println("Rectangle agrégé :");
        System.out.println(" - Point haut gauche : " + rectangle.getPointHG());
        System.out.println(" - Point bas droit : " + rectangle.getPointBD());
        System.out.println(" - Surface : " + rectangle.getSurface());
        System.out.println(" - Périmètre : " + rectangle.getPerimetre());
    }

}
```

# Classe ClasseTestImmuablesUtilitaire

```
public class ClasseTestImmuablesUtilitaire {  
    public static void afficherInstance(ClasseTestImmuables o) {  
        System.out.println(" -> attribut monString : \""  
            + o.getMonString()  
            + "\"\", attribut monInteger : "  
            + o.getMonInteger());  
    }  
}
```

note : la séquence \" indique que l'on veut afficher le caractère "

exemple de résultat :

-> attribut monString : "attribut chaîne modifiée", attribut monInteger : 120