

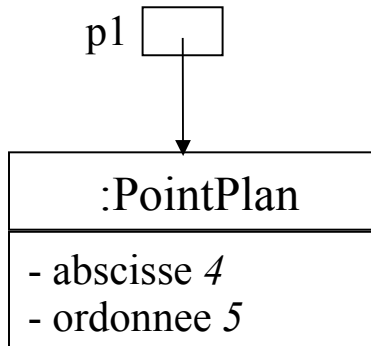
# Programmation objet (Cours 2)

1. Communiquer des objets à une méthode
2. Méthodes retournant un objet
3. Objets composés
4. Surcharge de méthodes/constructeurs
5. Chaînes de caractères

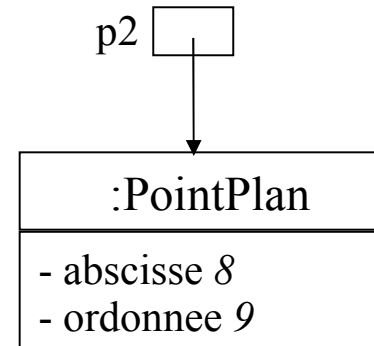
# 1) Communiquer des objets à une méthode

## Pour quoi faire ?

ex) `PointPlan p1 = new PointPlan(4, 5) ;`  
`p1.translate(10, 20) ;`



ex) `PointPlan p2 = new PointPlan(8, 9) ;`  
`float a = p2.getAbscisse() ;`



...mais une méthode peut avoir besoin d'autres objets en plus de l'objet courant

Ex : écrire une méthode comparant le point courant avec un autre point

# Comment communiquer des objets à une méthode ?

Fournir à la méthode la référence de chaque objet qu'elle utilise  
(en plus de l'objet courant référencé par **this**)

```
public boolean egaleA(PointPlan p)
```

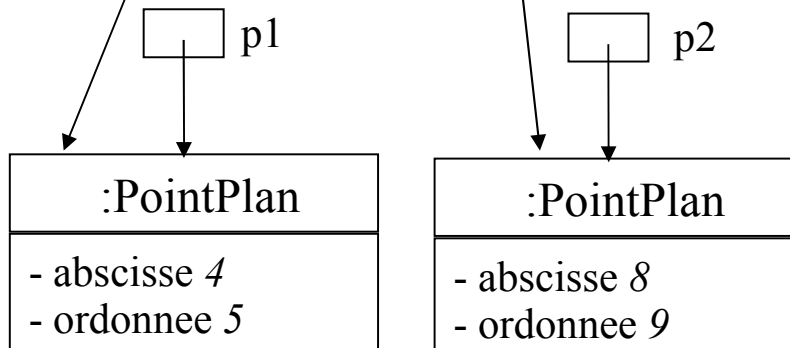
```
{
```

*p* reçoit la référence contenue dans *p2*  
(*this* reçoit la référence contenue dans *p1*)

```
    return
```

```
        (this.abscisse == p.abscisse &&  
         this.ordonnee == p.ordonnee);
```

```
}
```



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

```
{
```

```
    PointPlan p1 = new PointPlan(4, 5);
```

```
    PointPlan p2 = new PointPlan(8, 9);
```

```
    if ( p1.egaleA(p2))
```

```
        System.out.println("egaux");
```

```
    else
```

```
        System.out.println("pas egaux");
```

```
}
```

(attention : comparer les références  
*p1* et *p2* reviendrait à comparer les  
adresses et non les objets)

## 2) Méthodes retournant un objet

**Pourquoi faire ?** Le but d'une méthode peut être de :

1. créer et retourner un objet

ex) écrire une méthode *inverse()* qui retourne le point symétrique du point courant par rapport à l'origine

2. retourner plusieurs résultats

ex) écrire une méthode *coorPolaires()* qui retourne les coordonnées polaires du point courant

# Comment retourner un objet?

Créer un objet dans la méthode et retourner la référence de cet objet

```
public classe PointPlan  
{
```

précise que la méthode retourne une  
référence sur un *PointPlan*

```
public PointPlan inverse()  
{
```

```
    PointPlan p = new PointPlan( - this.abscisse,  
                                - this.ordonnee) ;
```

créé un point

```
    return p ;
```

```
}
```

retourne la  
référence du point  
créé

```
public static void main(String[] args)  
{  
    PointPlan p1 = new PointPlan(4, 5) ;  
    PointPlan p2 = p1.inverse() ;  
}
```

*p2* reçoit la  
référence fournie  
par la méthode  
*inverse*

:PointPlan

- abscisse -4  
- ordonnee -5

```
} // fin classe PointPlan
```

# Retourner un objet sans déclarer de référence

Créer un objet dans la méthode et retourner directement la référence  
fournie par **new**

```
public classe PointPlan
```

```
{  
    public PointPlan inverse()  
    {  
        return new PointPlan( - this.abscisse,  
                               - this.ordonnee) ;  
    }  
}
```

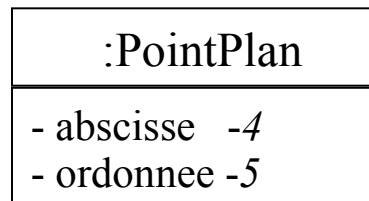
**new** crée un  
objet de type  
*PointPlan* et  
fournit sa  
référence

**return** retourne au point  
d'appel la référence  
fournie par **new**

```
} // fin classe PointPlan
```

```
public static void main(String[] args)  
{  
    PointPlan p1 = new PointPlan(4, 5) ;  
    PointPlan p2 = p1.inverse() ;  
}
```

*p2* reçoit la référence  
fournie par la méthode  
*inverse*



# Comment retourner plusieurs résultats ?

Créer un objet contenant les résultats et retourner sa référence

ex) retourner les coordonnées polaires (module et argument) d'un point du plan

**Solution ½ : définir la classe correspondant au type du résultat**

```
public class PointPolaire
{
    private double module ;
    private double argument ;
    public PointPolaire (double mod, double arg)
    {
        this.module = mod
        this.argument = arg ;
    }
    public double getModule()
    {
        return this.module ;
    }
    public double getArgument( )
    {
        return this.argument ;
    }
} // fin classe PointPolaire
```

les résultats seront  
stockés dans les  
variables d'instances  
*module* et *argument*

le constructeur initialise  
*module* et *argument* avec  
des valeurs fournies en  
paramètres

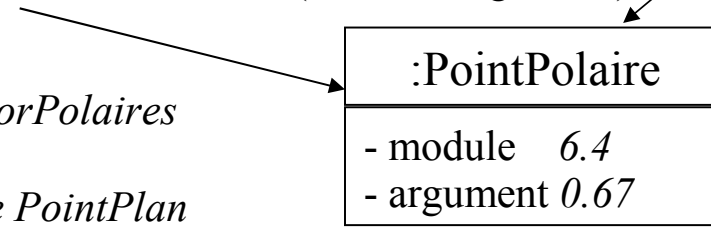
méthodes permettant  
d'accéder aux résultats

## Solution 2/2 : créer l'objet résultat et retourner sa référence

```
public class PointPlan
{
    ...
    // retourne les coordonnées polaires
    // du point courant
    public PointPolaire coorPolaires( )
    {
        double module = Math.sqrt(this.abscisse * this.abscisse +
                                   this.ordonnee * this.ordonnee) ;
        double argument = Math.atan(this.ordonnee / this.abscisse) ;

        return new PointPolaire(module, argument)

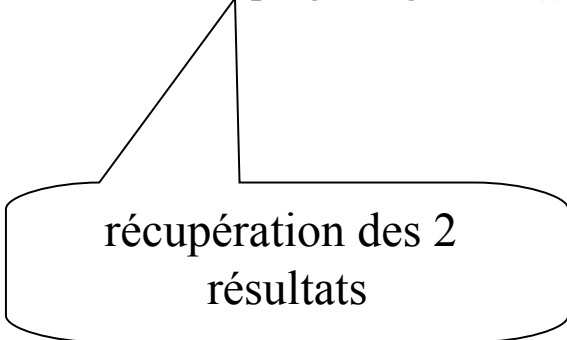
    } // fin coorPolaires
} // fin classe PointPlan
```



:PointPolaire
- module 6.4
- argument 0.67

```
public static void main(String[] args)
{
    PointPlan p1 = new PointPlan(4, 5) ;
    PointPolaire p2 = p1.coorPolaires() ;

    double m = p2.getModule( ) ;
    double a = p2.getArgument( ) ;
}
```

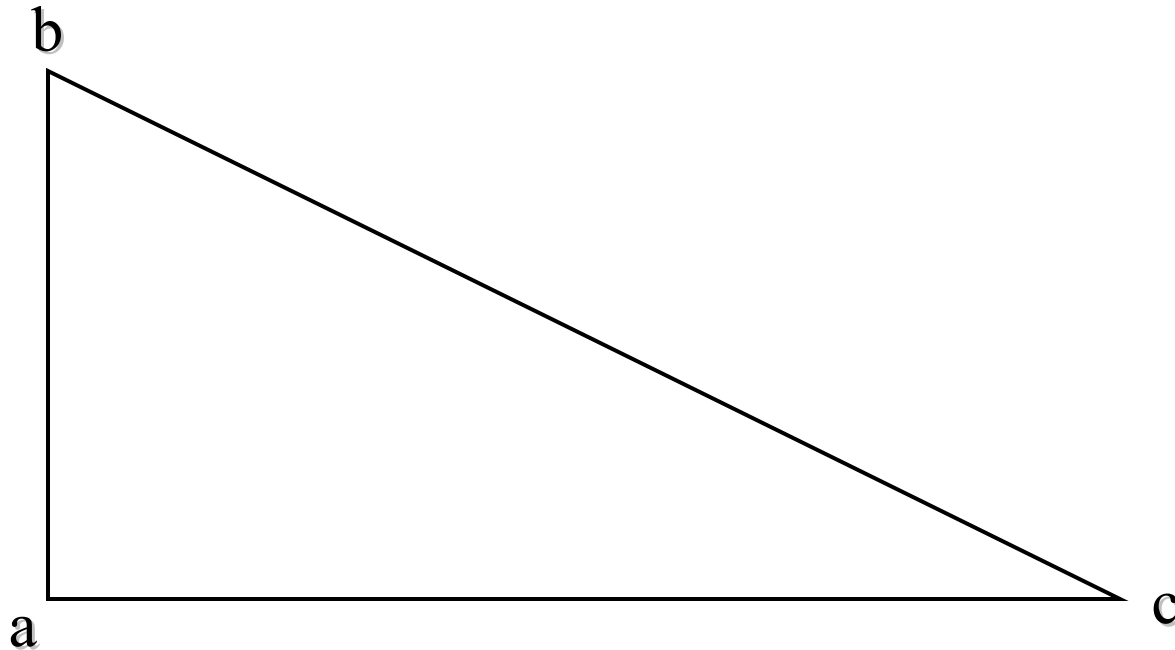


récupération des 2  
résultats



### 3) Créer des objets composés (objets contenant d'autres objets)

**Exemple 1/2 :** Un triangle est composé de 3 sommets

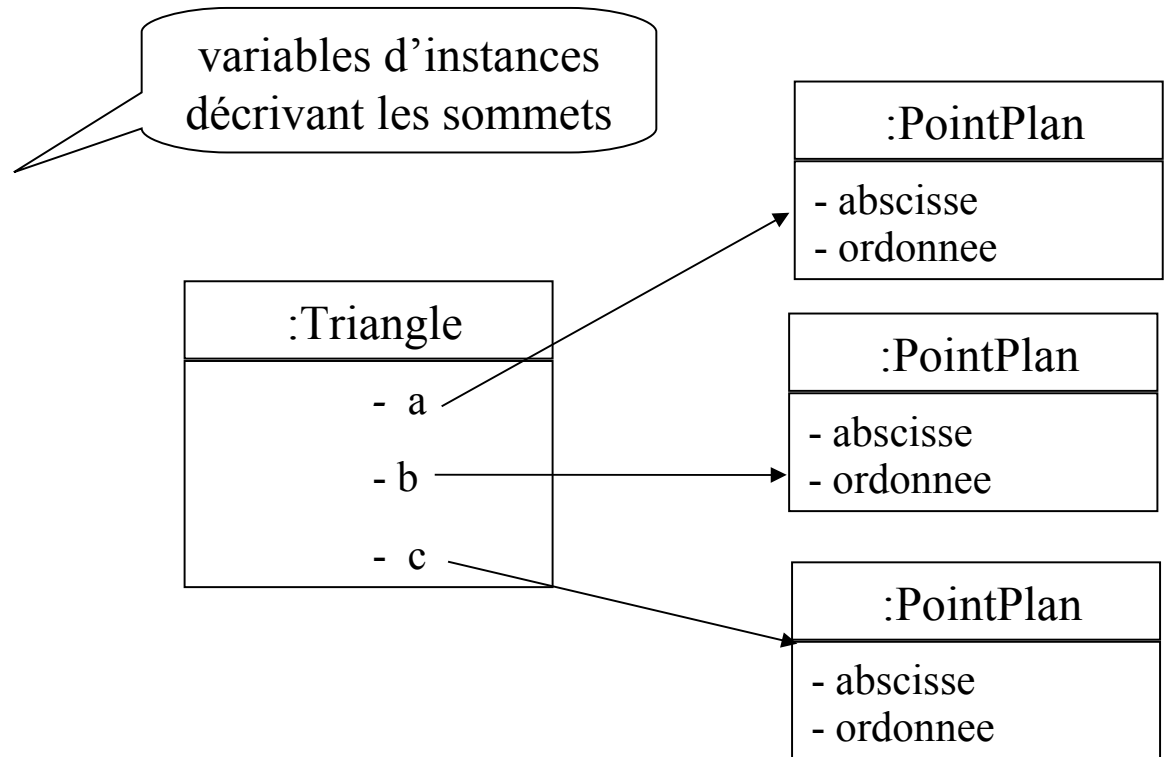


il faut définir 4 objets distincts :  
trois objets (*a*, *b* et *c*) de type *PointPlan* et un objet de type *Triangle*

## Exemple 2/2

```
public class Triangle
```

```
{  
    private PointPlan a ;  
    private PointPlan b ;  
    private PointPlan c ;  
  
    // constructeurs ...  
  
    public PointPlan getA()  
    {return this.a ;}  
  
    public PointPlan getB()  
    {return this.b ;}  
  
    public PointPlan getC()  
    {return this.c ;}  
}  
// fin classe Triangle
```



Les 4 objets doivent être créés et initialisés. Il y a 2 possibilités :

## a) Le triangle crée les sommets

```
public class Triangle
```

```
{  
    private PointPlan a ;  
    private PointPlan b ;  
    private PointPlan c ;
```

```
    public Triangle(float xa, float ya,  
                    float xb, float yb,  
                    float xc, float yc)
```

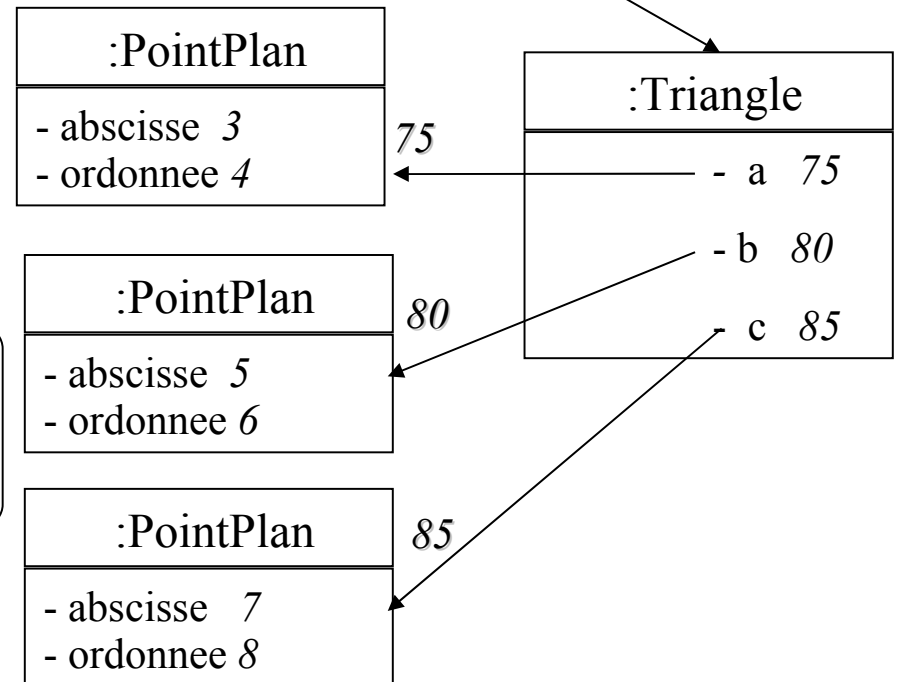
```
    this.a = new Point (xa, ya) ;  
    this.b = new Point (xb, yb) ;  
    this.c = new Point (xc, yc) ;
```

```
}
```

les variables-références désignant les sommets sont initialisées

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

```
{  
    Triangle t = new Triangle (3, 4, 5, 6, 7, 8) ;  
}
```



```
} // fin classe Triangle
```

## b) Les sommets sont d'abord créés en dehors du triangle

```
public class Triangle
```

```
{  
    private PointPlan a ;
```

```
    private PointPlan b ;
```

```
    private PointPlan c ;
```

```
    public Triangle(PointPlan pa, PointPlan pb, PointPlan pc)
```

```
{
```

```
    this.a = pa ;
```

```
    this.b = pb ;
```

```
    this.c = pc ;
```

```
}
```

2 – le triangle est créé (on lui fournit les références des sommets)

les variables-références désignant les sommets sont initialisées

```
} // fin classe Triangle
```

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

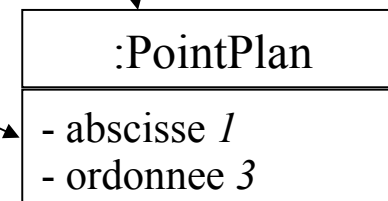
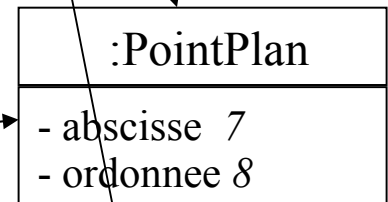
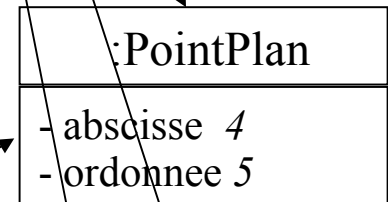
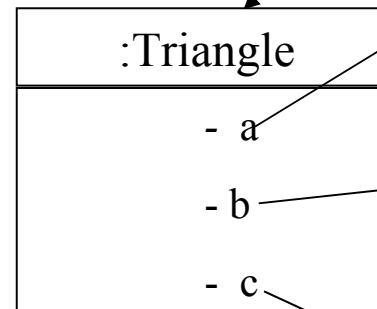
```
{
```

```
    PointPlan p1 = new PointPlan(3, 4) ;
```

```
    PointPlan p2 = new PointPlan(5, 6) ;
```

```
    PointPlan p3 = new PointPlan(7, 8) ;
```

```
    Triangle t = new Triangle (p1, p2, p3) ;
```

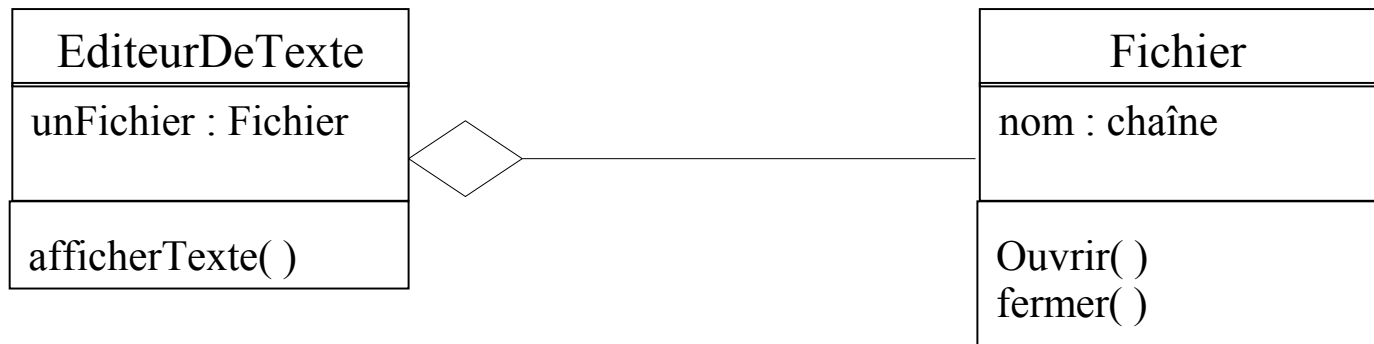


# Agrégation et composition

## Agrégation - Principe :

- un attribut d'une classe peut être un objet instance d'une autre classe
- l'élément agrégé (l'objet) peut être partagé entre différents éléments
- un élément peut exister indépendamment de son conteneur

## Représentation en UML :



# Agrégation et composition

- l'élément agrégé est référencé par un attribut dans le conteneur
- selon la cardinalité (nombre d'éléments agrégés), on utilisera soit une variable simple soit un tableau, une liste, ...

## En java

```
public class EditeurDeTexte{  
    ...  
    Fichier unFichier ;  
    ...  
    void AfficherTexte ( ) {  
        ...  
    }  
    ...  
}
```

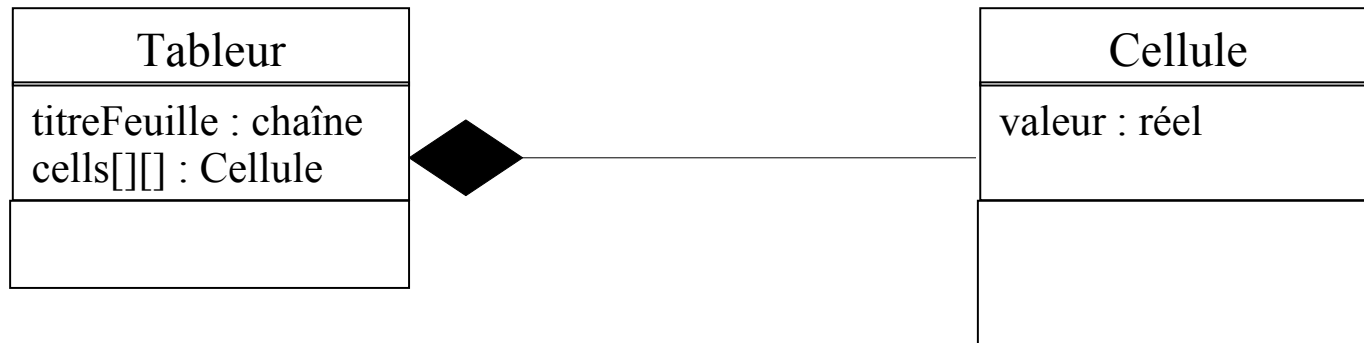
```
public class Fichier {  
    String nom ;  
    void Ouvrir () {  
        ...  
    }  
    void Fermer () {  
        ...  
    }  
    ...  
}
```

# Agrégation et composition

## Composition - Principe :

- la composition est proche de l'agrégation mais la relation est plus contraignante pour le contenu
- la suppression du conteneur entraîne la suppression du contenu
- un seul élément ne peut appartenir qu'à une seule composition

## Représentation en UML :



# Agrégation et composition

- créer les éléments composés dans le constructeur du conteneur
- la destruction du conteneur conduit a la destruction du contenu

## En java

```
public class Tableur {  
    ...  
    Cellule cells [ ][ ];  
    ...  
    void Tableur () {  
        // Instanciation des cellules  
    }  
    ...  
}
```

```
public class Cellule {  
    float valeur ;  
    ...  
}
```



## 4) Surcharger un(e) méthode/constructeur

Chaque méthode possède une signature

signature =

nom de la méthode,

+ nombre de paramètres,

+ type des paramètres

**public** boolean contientPoint (PointPlan p)



le type de retour et le modificateur d'accès (**public**, **private**) ne sont pas caractéristiques de la signature

# Surcharge d'une méthode

Une méthode est surchargée lorsqu'elle existe en plusieurs exemplaires de signatures différentes au sein d'une classe

```
public class PointPlan
{
    // renvoie true si le point courant a mêmes abscisse et ordonnée que le point p
    public boolean egaleA(PointPlan p)
    {
        return
            (this.abscisse == p.abscisse &&
             this.ordonnee == p.ordonnee) ;
    }

    // renvoie true si le point courant a mêmes abscisse et ordonnée que
    // le point de coordonnées (x, y)
    public boolean egaleA(float x, float y)
    {
        return
            (this.abscisse == x &&
             this.ordonnee == y) ;
    }
}

// fin classe PointPlan
```

# Surcharge de constructeurs

Une classe peut contenir plusieurs constructeurs de signatures différentes

```
public class PointPlan
{
    public PointPlan(float x, float y)
    {
        this.abscisse = x ;
        this.ordonnee = y ;
    }
}
```

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

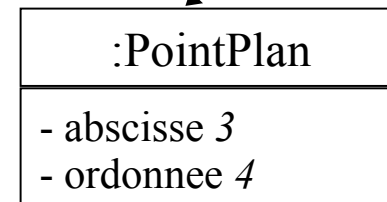
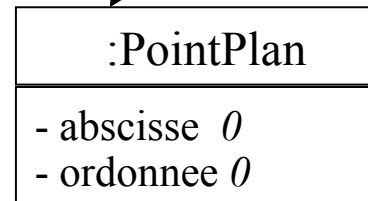
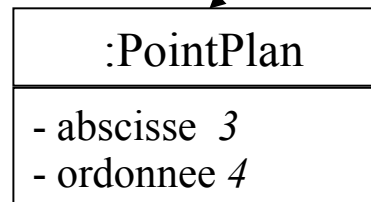
```
{ PointPlan p1 = new PointPlan(3, 4) ;
```

```
PointPlan p2 = new PointPlan( ) ;
```

```
PointPlan p3 = new PointPlan(p1) ;
```

```
public PointPlan( )
```

```
{
    this.abscisse = 0 ;
    this.ordonnee = 0 ;
}
```



```
public PointPlan(PointPlan p )
{
    this.abscisse = p.abscisse ;
    this.ordonnee = p.ordonnee ;
}
```

```
} // fin classe PointPlan
```

Constructeur par copie : on initialise l'objet courant avec le contenu d'un objet de même classe passé en argument

# Un constructeur peut en appeler un autre

Un constructeur peut invoquer (appeler) un autre constructeur de la même classe en utilisant *this(...)* en 1ère instruction

```
public class PointPlan
{
    public PointPlan(float x, float y)
    {
        this.abscisse = x ;
        this.ordonnee = y ;
    }
}
```

important : ne pas confondre

la référence *this* et l'invocation de constructeur *this()*.

```
public PointPlan()
{
    this(0 , 0) ;
}
```

appelle le constructeur *PointPlan(float x, float y)*  
avec les paramètres effectifs (0 , 0).

```
} // fin classe PointPlan
```

Grâce à *this()* seul le constructeur de signature  
*PointPlan(float x, float y)* effectue les initialisations  
(et éventuellement les contrôles sur la validité des  
valeurs)

## **5) Chaînes de caractères**

### **Chaînes de caractères :**

Une chaîne de caractères est une suite ordonnée (séquence) de caractères

ex) "bonjour" (7 caractères)

ex) "" (chaîne vide : 0 caractère)

# La classe String

Assure la gestion des chaînes de caractères

## Constructor Summary

### **String**(String s)

Initialise un nouvel objet de type *String* avec la séquence de caractères contenue dans l'objet référencé par *s* (constructeur par copie)

## Method Summary

boolean

### **equals**(String s )

Retourne *true* si la chaîne courante (référéncée par *this*) et la chaîne référencée par *s* possède la même séquence de caractères, sinon *false*.

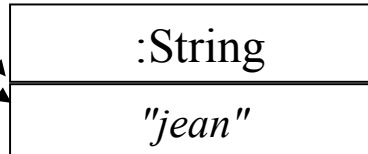
int

### **length**( )

Retourne le nombre de caractères contenus dans la chaîne courante (0 si la chaîne est vide)

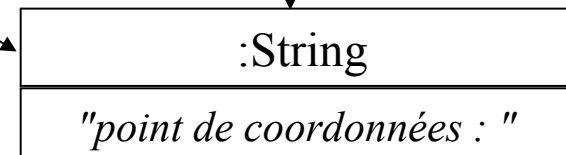
# Examples

```
{  
    System.out.println("bonjour") ;  
  
    String s = "jean" ;  
}
```

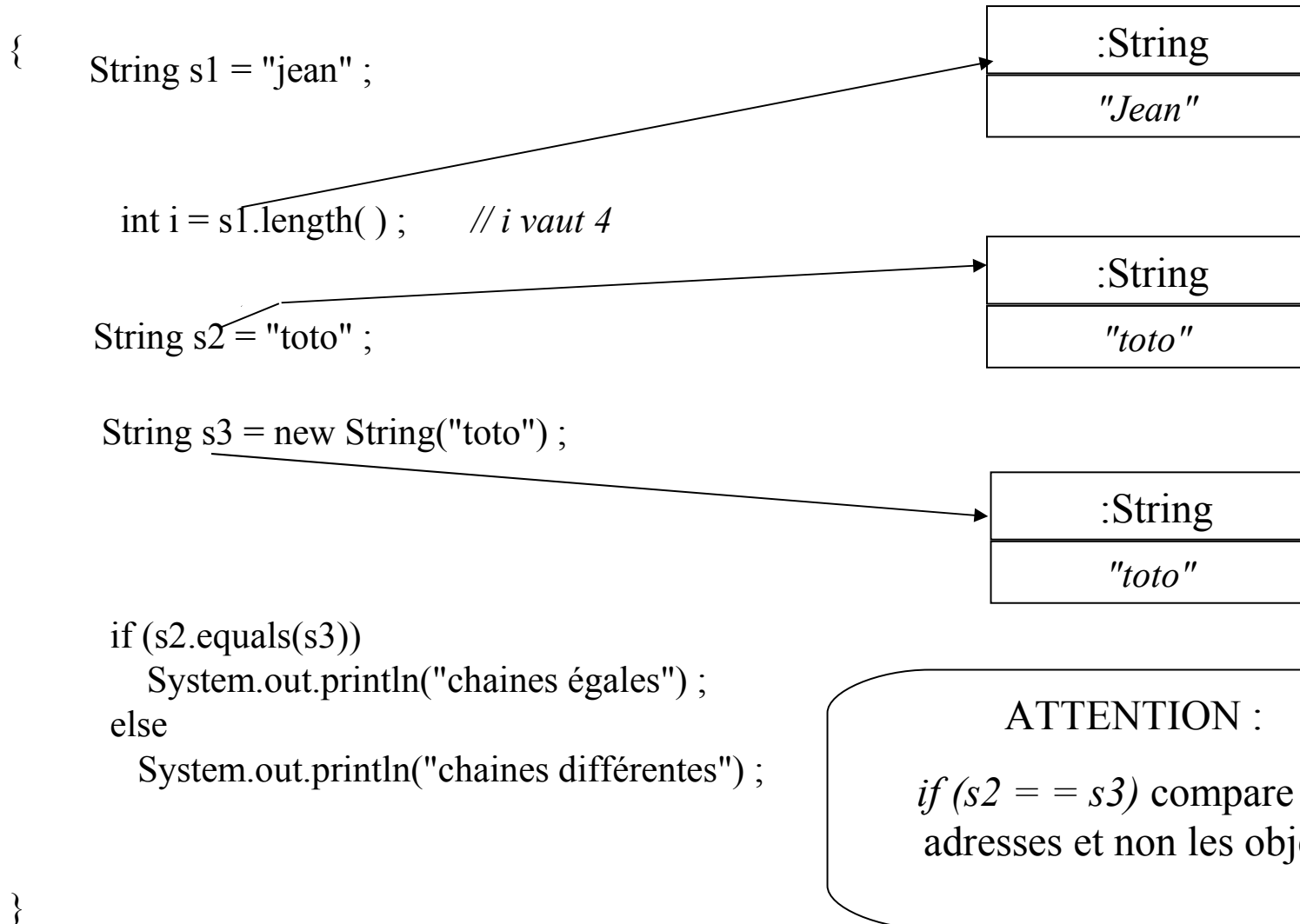


```
public class PointPlan  
{  
    ...  
    public affiche(String message)  
    {  
        System.out.print(message) ;  
        System.out.print(this.abscisse) ;  
        System.out.print(this.ordonnee) ;  
    }  
} // fin classe PointPlan
```

```
public static void main(String[] args)  
{  
    PointPlan p = new PointPlan(3, 4) ;  
    p.affiche("point de coordonnees : ") ;  
}
```



# Exemples d'utilisation de chaînes



ATTENTION :

*if (s2 == s3) compare les  
adresses et non les objets*

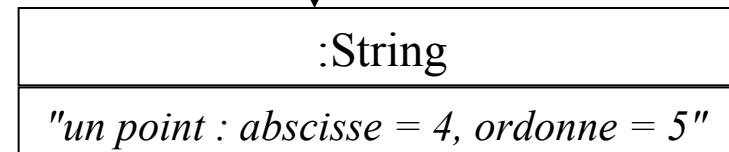
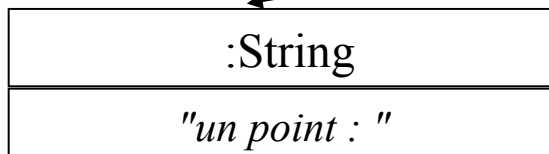
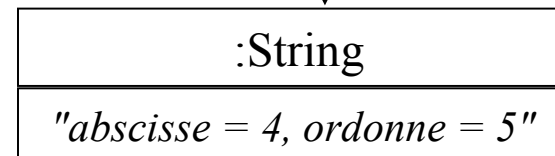


# La méthode toString()

*toString()* est implicitement appelée lorsqu'une variable-référence apparaît en argument de *System.out.println(...)* ou dans une concaténation

ex) {  
    PointPlan p = new PointPlan(4,5) ;  
    System.out.println(p) ;   ↔   System.out.println(p.toString()) ;  
}

ex) {  
    PointPlan p = new PointPlan(4,5) ;  
    String s = "un point : " + p ;   ↔   "un point : " + p.toString() ;  
}

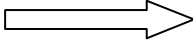


## Sans redefinition de *toString()*

La méthode *toString()* de la classe *Object* fournit une chaîne contenant :

- le nom de la classe concernée,
- l'adresse de l'objet en hexadécimal (précédée de @)

```
ex) {  
    PointPlan p = new PointPlan(4,5) ;  
    System.out.println(p) ;  
}
```



PointPlan@fd16adcf

# La méthode *toString()*

Elle doit être redéfinie dans chaque classe pour qu'elle retourne la chaîne de caractères représentant le contenu d'un objet.

```
public class PointPlan
{
    this.abscisse ;
    this.ordonnee ;
    ...
    public String toString( )
    {
        return ("abscisse = " + this.abscisse +
                ", ordonnee = " + this.ordonnee) ;
    }
} // fin classe PointPlan
```

```
public static void main(String[] args)
{
    PointPlan p = new PointPlan(4,5) ;
    String s = p.toString( ) ;
}
```

:String

*"abscisse = 4, ordonnee = 5"*