

# **Introduction à la programmation objet (Cours 1)**

1. Classes, instances et références
2. Vue détaillée d'une classe
3. Encapsulation
4. Créer la documentation d'un programme Java

# Java en bref

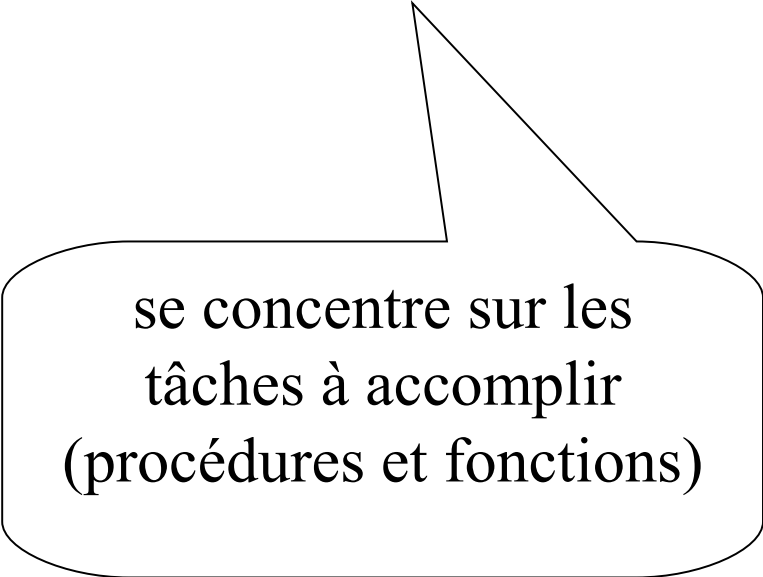
Diffusé par Sun en 1995, Java est un langage:

- o Orienté objet
- o Portable
- o Adapté à Internet
- o Multi-tâches

# Java est un langage orienté objet

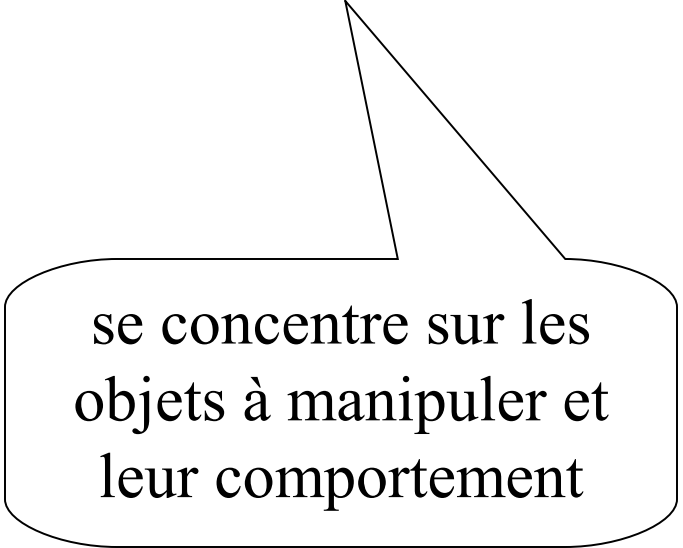
Pour résoudre un problème:

La programmation procédurale / La programmation objet



se concentre sur les  
tâches à accomplir  
(procédures et fonctions)

A speech bubble pointing from the text 'La programmation procédurale' to this box.



se concentre sur les  
objets à manipuler et  
leur comportement

A speech bubble pointing from the text 'La programmation objet' to this box.

# Java est un langage portable

Tout programme java peut s'exécuter sans modification dans différents environnements logiciels et matériels

- Windows, Linux, Mac Os
- Pentium, AMD
- ...

# Java est un langage adapté à Internet

- Des programmes java (*applets*) peuvent être intégrés dans des documents HTML (qui constituent les pages Web)
- Java gère les protocoles de communication utilisés par internet (TCP/IP, http, FTP...)

# **Java est un langage multitâches**

Java dispose des mécanismes qui permettent de créer et de synchroniser des tâches (programmes) qui s'exécutant simultanément

# 1. Classes, instances (objets) et références

## Classe :

Une classe est un **modèle** d'objet : elle décrit un ensemble de caractéristiques (données) et de comportements (méthodes)

Ce modèle est utilisé pour construire des objets ayant ces caractéristiques et ces comportements

# Exemple

```
public class PointPlan
```

Le nom de la classe commence par une Majuscule

```
{
```

```
    private float abscisse ;
```

```
    private float ordonnee ;
```

```
    // initialise le point avec (x, y)
```

```
    public PointPlan(float x, float y)
```

```
{
```

```
        this.abscisse = x ;
```

```
        this.ordonnee = y ;
```

```
}
```

```
    // translate le point de dx et dy
```

```
    public void translate(float dx, float dy)
```

```
{
```

```
        this.abscisse = this.abscisse + dx ;
```

```
        this.ordonnee = this.ordonnee + dy ;
```

```
}
```

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

Donnée(s)  
(variables  
d'instances)

Constructeur(s)

Méthode(s)  
(d'instances)

PointPlan
- float abscisse
- float ordonnee
+ PointPlan( )
+ void translater( )

(UML)



# Instance :

Une instance est un objet concret créé à partir d'une classe pendant l'exécution d'un programme.

Une instance possède un espace mémoire propre.

(Instance = objet)

# Exemple d'instanciation

la méthode *main* est le point d'entrée dans la classe (explicité plus loin)

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

```
{
```

```
    PointPlan p1 ;
```

```
    PointPlan p2 ;
```

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

appel obligatoire du constructeur *PointPlan*  
qui initialise l'instance

```
    p2 = new PointPlan(7, 9) ;
```

```
}
```

chaque instance possède ses propres variables (d'instances).

:PointPlan

- abscisse 4  
- ordonnee 5

:PointPlan

- abscisse 7  
- ordonnee 9

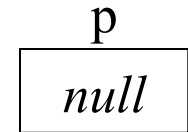
# Variable de type référence :

Une variable de type référence (*variable-référence*) est une variable contenant la référence (l'adresse) d'une instance

{ // début bloc d'instructions d'une méthode Java

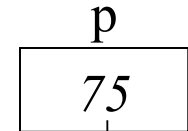
PointPlan p ;

Déclaration d'une variable-référence *p* sur une (future) instance de *PointPlan*

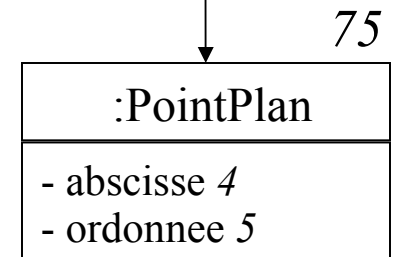


p = new PointPlan(4, 5) ;

*p* reçoit l'adresse (fournie par **new**) de l'instance de *PointPlan* venant d'être créée



} // fin bloc d'instructions



# Synthèse

```
PointPlan p ;
```

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

- **new** réserve l'espace mémoire pour une instance de type *PointPlan* (création de l'instance)
- *PointPlan(4,5)* initialise l'instance créée
- **new** retourne l'adresse de l'instance créée qui est affectée à la variable-référence *p*

# Usage des variables-références (1/3)

On ne peut accéder à une instance qu'au moyen d'une variable référence

{

Déclaration et initialisation de la variable-référence *p*  
avec l'adresse d'une instance de *PointPlan*

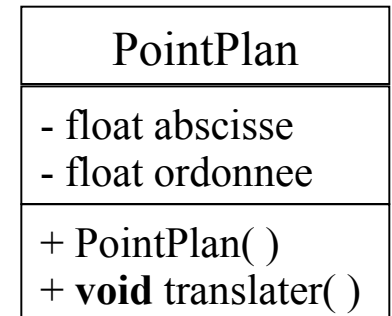
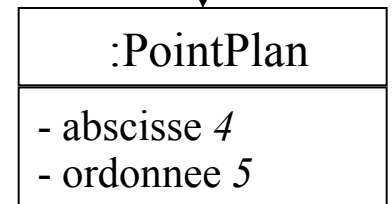
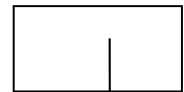
```
PointPlan p = new PointPlan(4, 5);
```

```
p.translater(10, 20);
```

}

appel (activation) de la  
méthode *translater* qui  
s'applique à l'instance  
référéncée par *p*

p

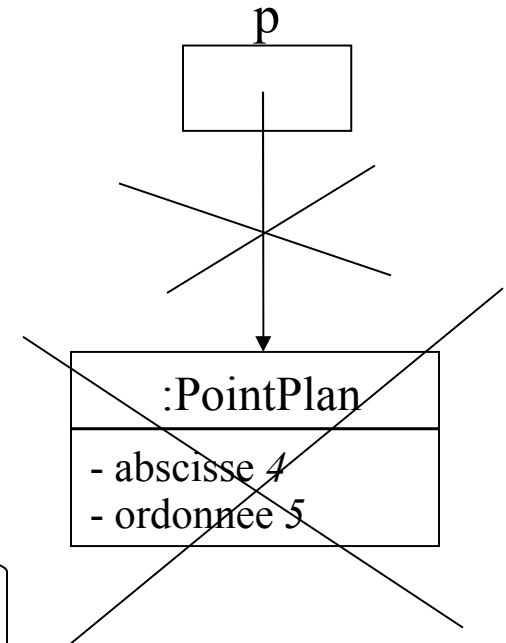


# Usage des variables-références (2/3)

Une instance qui n'est plus référencée est inaccessible

```
{  
    PointPlan p = new PointPlan(4, 5);  
  
    p = null;  
  
    p.translater(10, 20);  
}
```

Engendre une erreur à l'exécution car aucune instance n'est associée à la variable-référence *p*

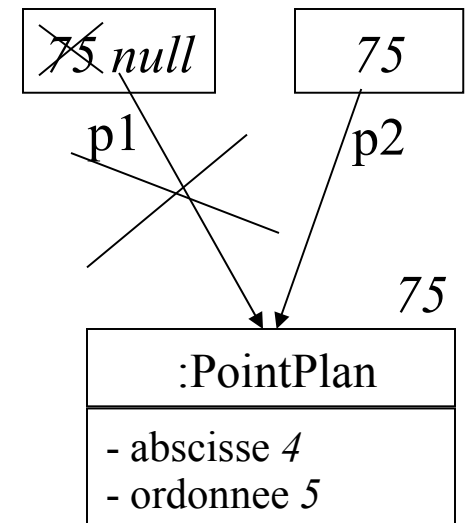


Tout instance non référencée est éliminée par le *Garbage Collector* (ramasse-miettes)

# Usage des variables-références (3/3)

Une instance peut être référencée par plusieurs variables-références

```
{  
    PointPlan p1 = new PointPlan(4, 5) ;  
    PointPlan p2 = p1 ;  
    p1 = null;  
    p2.translater(10, 20) ;  
}
```



Aucune erreur à l'exécution: une instance de *PointPlan* est bien associée à la variable-référence *p1*

## 2. Structure d'une classe

```
public class uneClasse
```

```
{
```

Variables d'instances

Définissent ce qu'est l'objet

Constructeurs

Initialisent l'objet à sa  
création

Méthodes

Définissent le comportement  
(les fonctions) de l'objet

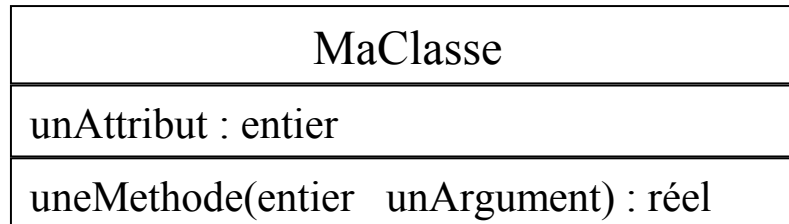
```
} // fin classe uneClasse
```



# Représentation en UML

une classe est représentée par une boîte à trois compartiments :

- le nom de la classe,
- les attributs de la classe (sous la forme nom : type),
- les méthodes de la classe (sous la forme nom(type nom, ...) : type de retour



(UML)

# Variables d'instances

On distingue les variables de type primitif (variables classiques qui ne représentent pas un objet) et les variables-références

```
public class PointPlan
{
    private float abscisse ;

    private float ordonnee ;

    private String nomPoint ;
}
```

Variable-référence sur une instance  
de la classe String (vu au prochain  
cours)

types primitifs :  
byte (entier 8 bits)  
short (entier 16 bits)  
int (entier 32 bits)  
long (entier 64 bits)  
float (réel 32 bits)  
double (réel 64 bits)  
char (caractère, 'A', '2', ...)  
boolean (true, false)

Les variables d'instances sont utilisables dans toutes les méthodes d'instances de la classe

# Constructeur

Un constructeur est une méthode particulière appelée par l'opérateur **new** au moment où une instance est créée

```
public class PointPlan
{
    private float abscisse ;
    private float ordonnee ;

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

    public PointPlan( )
    {
        this.abscisse = 0 ;
        this.ordonnee = 0 ;
    }
} // fin classe PointPlan
```

le nom d'un constructeur est celui de la classe.  
Un constructeur ne retourne jamais de valeur:  
aucun type ni **void** ne lui est associé

Il peut y avoir plusieurs constructeurs (au moins 1) qui diffèrent par leur signature (nombre et type des arguments).

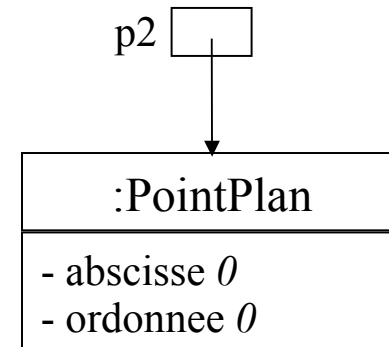
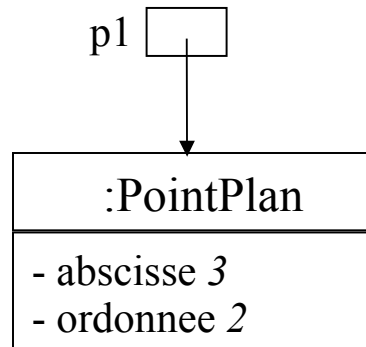
Un constructeur sert (en général) à initialiser les variables de l'instance venant d'être créée

# Exemple d'utilisation de constructeurs

```
public class PointPlan
```

```
{  
    private float abscisse ;  
    private float ordonnee ;  
  
    public PointPlan(float x, float y)  
    {  
        this.abscisse = x ;  
        this.ordonnee = y ;  
    }  
  
    public PointPlan( )  
    {  
        this.abscisse = 0 ;  
        this.ordonnee = 0 ;  
    }  
}
```

```
public static void main(String[ ] args)  
{  
    PointPlan p1 = new PointPlan(3, 2) ;  
    PointPlan p2 = new PointPlan( ) ;  
} // fin main  
}  
// fin classe PointPlan
```



# Méthodes d'instances

Une méthode d'instance d'une classe est une fonction qui s'applique à une instance de cette classe.

```
public class PointPlan
```

```
{  
    .....
```

```
    public void translate(float dx, float dy)
```

```
{  
    this.abscisse = this.abscisse + dx ;  
    this.ordonnee = this.ordonnee + dy ;  
}
```

```
    public double distance()
```

```
// retourne la distance du point à l'origine
```

```
{  
    float d ;  
    d = this.abscisse * this.abscisse +  
        this.ordonnee * this.ordonnee ;  
    return Math.sqrt(d) ;  
}
```

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

une méthode d'instance peut avoir des arguments (paramètres formels) de type primitif ou de type référence

une méthode d'instance peut retourner une valeur de type primitif, ou une référence (vu au prochain cours), ou ne rien retourner (**void**)

une méthode d'instance peut déclarer et utiliser des variables locales (de type primitif ou de type référence)

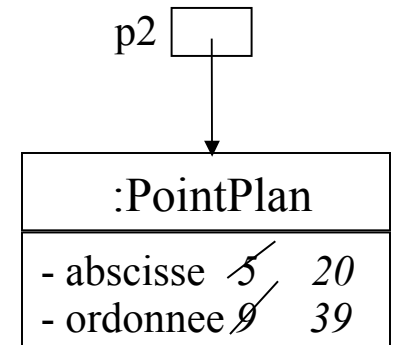
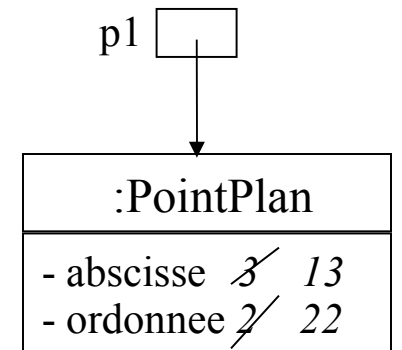
Expliqué plus tard

# Appel des méthodes d'instances

Une méthode d'instance est activée au moyen d'une variable référence et s'applique à l'instance désignée par cette référence

```
public static void main(String[ ] args)
{
    PointPlan p1 = new PointPlan(3, 2) ;
    p1.translate(10, 20) ;
    PointPlan p2 = new PointPlan(5, 9) ;
}
```

En Java on peut déclarer des variables n'importe où au sein d'un bloc d'instructions {...}. Une variable n'est utilisable que dans le bloc où elle est déclarée



# En java les paramètres sont passés par valeur

chaque paramètre effectif fournit sa valeur au paramètre formel  
correspondant dans la méthode appelée (comme en C)

les paramètres formels sont locaux à la  
méthode où ils sont déclarés.

```
public static void main(String[ ] args)
{
    PointPlan p1 = new PointPlan(3, 2);
    float tx = 10;
    p1.translate(tx, 20);
}
```

```
public void translate (float dx, float dy)
{
    this.abscisse = this.abscisse + dx;
    this.ordonnee = this.ordonnee + dy;
}
```

Le principe est identique pour les variables-références  
(vu au prochain cours)

# this

Quand on écrit une méthode d'instance on ne sait pas sur quelles instances elle s'appliquera (les instances sont créées par **new** en cours d'exécution).

**this** contient la référence de l'instance sur laquelle s'applique la méthode appelée

```
public void translate(float dx, float dy)
```

```
{  
    this.abscisse = this.abscisse + dx ;  
    this.ordonnee = this.ordonnee + dy ;  
}
```

**this** == p2 pour l'appel  
p2.translate(15, 30)

p2

:PointPlan

- abscisse 5  
- ordonnee 9

**this** == p1 pour l'appel  
p1.translate(10, 20)

p1

:PointPlan

- abscisse 3  
- ordonnee 2

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

```
{  
    PointPlan p1 = new PointPlan(3, 2) ;  
    PointPlan p2 = new PointPlan(5, 9) ;  
    p1.translate (10, 20) ;  
    p2.translate(15, 30) ;  
}
```



### 3. Encapsulation

Une classe contient une partie publique qui décrit les services qu'elle offre et une partie privée à usage interne

Le mot clé **private** indique que la donnée ou la méthode concernée est inaccessible (et invisible) depuis l'extérieur de la classe

Le mot clé **public** signale qu'il n'y a aucune restriction

# Exemple (1/2)

```
public class PointPlan
{
    private float abscisse ;
    private float ordonnee ;
    public int couleurPoint ; // déconseillé !

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

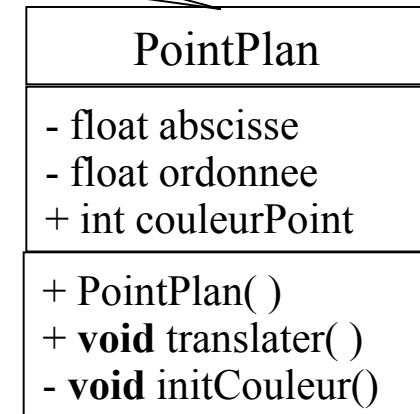
    public void translate(float dx, float dy)
    {
        this.abscisse = this.abscisse + dx ;
        this.ordonnee = this.ordonnee + dy ;
    }

    private void initialiseCouleur( )
    {
        // methode à usage interne

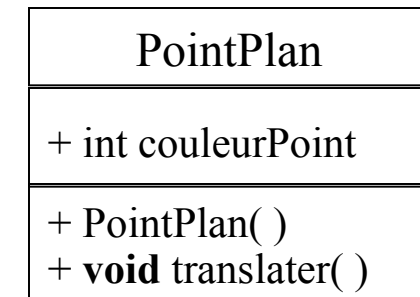
    } ...

} // fin classe PointPlan
```

vue privée (interne)



vue publique (externe)



# Exemple (2/2)

```
public class PointPlan
{
    private float abscisse ;
    private float ordonnee ;
    public int couleurPoint ;

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

    public void translate(float dx, float dy)
    {
        this.abscisse = this.abscisse + dx ;
        this.ordonnee = this.ordonnee + dy ;
    }

    private void initialiseCouleur( )
    { // methode à usage interne

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

```
public class Test // autre classe
{
    ...
    public static void main (String[] args)
    {
        PointPlan p = new PointPlan(7,5) ;

        p.abscisse = 4 ;
        p.couleurPoint = 2 ;
        p.initialiseCouleur( ) ;
        p.translate( ) ;

    } // fin main
} // fin classe Test
```

rejet du compilateur :  
*abscisse* est inconnu  
(car privé)

ok: *couleurPoint* est  
connu (car publique)

rejet du compilateur :  
méthode inconnue

ok

# Objectif de l'encapsulation

- Simplifier la tâche de l'utilisateur d'une classe en lui masquant les détails internes
- Protéger le contenu des objets (impossible d'accéder directement aux variables privés ou d'activer des méthodes privées)
- Rendre l'utilisation d'une classe indépendante de sa structure interne (quand la partie privée change la partie publique demeure identique)

# Une technique d'encapsulation sûre

Déclarer toutes les variables d'instances en accès privé (**private**)

Pour les variables dont on veut autoriser un accès :

- en lecture : définir une méthode publique retournant la valeur de la variable
- en écriture : définir une méthode publique affectant la variable avec une nouvelle valeur passée en argument (on pourra ainsi contrôler la valeur fournie)

```
public class PointPlan
{
    private float abscisse ;

    public float getAbscisse() // lecture
    {
        return this.abscisse ;
    }

    public void setAbscisse(float valeur) // écriture
    {
        this.abscisse = valeur ;
    }
}
```

```
public class Test
{
    public static void main (String[] args)
    {
        PointPlan p = new PointPlan(7, 5) ;

        float a = p.getAbscisse() ;

        p.setAbscisse(15) ;
    }
}
```

# La visibilité

- La visibilité d'un élément/membre (méthode ou attribut) est définie lors de la déclaration
- Quatre visibilités sont possibles
- On précède la déclaration d'un mot clé parmi : `private`, `protected` et `public`

**private (-)** : même classe

**protected(#)** : même classe, classe filles, même paquetage

**public(+)** : même classe, classe filles, même paquetage, autre cas

aucun (« friendly ») **( )** : même classe, même paquetage

# La visibilité

## Représentation en UML et déclaration en java

```
public class UneClasse {  
    private int unAttributPrive ;  
    protected float unAttributProtege ;  
    public String unAttributPublic ;  
    char unAttributFriendly ;  
    private float UneMethodePrivee () {  
        ...  
    }  
    protected int UneMethodeProtegee () {  
        ...  
    }  
    public float UneMethodePublique (int anInt) {  
        ...  
    }  
    String UneMethodeFriendly () {  
        ...  
    }  
}
```

UneClasse
- unAttributPrive : entier # unAttributProtege : réel + unAttributPublic : chaîne unAttributFriendly : caractère
- uneMethodePrivée ( ) : réel # uneMethodeProtégée ( ) : entier + uneMethodePublique (entier unEntier ) : réel uneMethodeFriendly ( ) : chaîne

# Documenter un programme (Javadoc)

- **Javadoc** est un outil permettant de générer la documentation des classes au format html.
- Toute la documentation des classes java (<https://docs.oracle.com/javase/8/docs/api/index.html>) est générée grâce à ce mécanisme.
- Vous devez prendre l'habitude d'avoir ce lien ouvert lorsque vous développez afin de vous documenter sur les classes et les méthodes que vous manipulez.



# Documenter un programme (Javadoc)

- La documentation est générée grâce à des commentaires spéciaux renseignés tout au long du code.
- Voici quelques mots clés utilisés :
  - Les commentaires sont entourés par `/** ... */`
  - Chaque ligne de commentaire commence par `*`
  - Chaque mot clé commence par `@`
  - `@version` : indique la version de la classe ou de la méthode
  - `@author` : indique le ou les auteurs
  - `@see` : renvoie vers d'autres références
  - ...

# Exemple – Javadoc

```
/**  
 * gestion d'un point du plan  
 * @version 1.0  
 * @author M Hakem  
 */
```

```
public class PointPlan
```

```
{
```

```
    private float abscisse ;
```

```
    private float ordonnee ;
```

```
    /**
```

```
     * initialise un point: x est l'abscisse, y l'ordonnee
```

```
     */
```

```
    public PointPlan(float x, float y)
```

```
    {
```

```
        this.abscisse = x ;
```

```
        this.ordonnee = y ;
```

```
    } // fin constructeur
```

```
    /**
```

```
     * decale le point courant de dx (horiz.) et dy (verti.)
```

```
     */
```

```
    public void translate(float dx, float dy)
```

```
    {
```

```
        this.abscisse = this.abscisse + dx ;
```

```
        this.ordonnee = this.ordonnee + dy ;
```

```
    } // fin methode translate
```

```
    /**
```

```
     * retourne la distance du point courant a l'origine
```

```
     */
```

```
    public double distance()
```

```
    {
```

```
        return Math.sqrt(this.abscisse * this.abscisse +  
                        this.ordonnee * this.ordonnee) ;
```

```
    } // fin methode distance
```

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

# Documentation html

javadoc *PointPlan.java* ↵

---

public class **PointPlan**

gestion d'un point du plan

**Version:**

1.0

**Author:**

M Hakem

## Constructor Summary

**PointPlan**(fl oat x, fl oat y)

initialise un point: x est l'abscisse, y l'ordonnee

## Method Summary

void	<b>affiche</b> ()
	affiche les coordonnees du point courant
double	<b>distance</b> ()
	retourne la distance du point courant a l'origine
void	<b>translate</b> (fl oat dx, fl oat dy)
	decale le point courant de dx (horiz.) et dy (verti.)