Remise à niveau Programmation

Java – Classes – Objets

Types atomique et variables référence

Javadocs

#1

Licence Pro
Métiers de l'Informatique : Conception, Développement,
Test de Logiciels, Parcours "Génie Logiciel, Système d'Information"
2020-2021

Présentation du langage Java

Un langage de programmation

- Implémentation du paradigme objet :
  - Organisation du code centrée sur les données à manipuler.
  - Par opposition aux paradigmes procédural ou fonctionnel qui sont plutôt centrés sur les procédures et leur enchainement.
- Langage centré sur les Objets qui sont structurés par :
  - Des attributs qui définissent l'objet et son état par un ensemble de données (valeurs/variables)
  - Des méthodes qui décrivent ce que peut faire l'objet (actions sur ses propres données ou sur les données d'autre(s) objet(s) passé(s) en paramètre)
- Syntaxe dérivée du C++
- Types primitifs semblables : **char** (16bits), **byte** (8bits), **short** (16bits), **int** (32bits), **long** (64bits), **float** (32bits), **double**, (64bits), **boolean** (8bits)
- Pas de type string (remplacé par une classe **`String`**).
- Mêmes opérateurs de comparaison et mêmes opérateurs booléens
- Des structures de contrôle similaires :

<table>
<tr>
<td>

```
if ( cond_1 ) {
    actionSI_1 // code
}
else if (cond_2 ){
    actionSI_2 // code
}
else {
    actionSINON // code
}
```

</td>
<td>

```
switch(variable){
   case 'val1':
         action1;
         break;
   case 'val2':
        'val3':
         action23;
         break;
   default:
         actionDef;
         break;
}
```

</td>
</tr>
<tr>
<td>

```
while  ( cond ) {
    action // code
}
```

</td>
<td>

```
for ( int i = 0 ; i<10 ; i++) {
    action // code
}
```

</td>
</tr>
<tr>
<td>

```
do {
    action // code
} while ( cond ) ;
```

</td>
<td>

```
for ( PointPlan p : points) {
    action // code
}
```

</td>
</tr>
</table>

Un environnement de programmation

- Un compilateur générant un bytecode mutliplateforme (class, jar)
- Une machine virtuelle sur la plateforme d'exécution (JVM ou JRE)
- API puissante (nombreuses librairies pour de nombreux services)
- javadoc : génération de documentation
- IDE : Eclipse
- JUnit : implémentation des tests unitaires

Un premier Programme

Le code source

```
// Fichier Hello.java

public class Hello
{
     public static void main( String [] args )
     {
          System.out.println( "Hello world") ;
     }
}
```

Commande de compilation

```
login@bash> javac Hello.java
```

Génération de **`Hello.class`** qui est un fichier en bytecode (binaire d'exécution).
Le nom de fichier est conservé mais l'extension change de **`.java`** pour **`.class`** .

Commande d'exécution

```
login@bash> java Hello
```

Cette commande permet d'exécuter le bytecode du fichier **`Hello.class`**.
Ce code est exécutable sur n'importe quelle plateforme disposant de la machine virtuelle.

Le fichier contenant le bytecode a exécuter présente nécessairement l'extension **`.class`**.
Cette extension **`.class`** est systématiquement dans l'appel en ligne de commande

Remarque :

- extension du fichier source est **`.java`**
- extension (automatique) du bytecode : **`.class`**
- exécution du bytecode : omission de l'extension **`.class`** lors de l'appel

Remarque :

- Tout code est déclaré dans une classe.
- Toute classe est déclarée dans un fichier qui porte le **`NomDeLaClasse.java`**
- Le programme principal (le seul qui peut être appelé par le système) est une méthode appelée **`main`** qui est publique et statique (nous reviendrons sur ces termes plus tard)
    - Tjrs : **`public static void main( String [] args) { …}`**
    - La méthode est déclarée dans une classe dédiée portant en général le nom du programme (ici **`Hello`**)

Définition des classes

Les classes

- C'est un modèle (dans un premier temps on peut dire qu'il s'agit d'une définition) sur la base de laquelle vont être construit (instancié) des objets
- C'est un type (comme en C) auquel on associe fortement des définitions de fonctions : les méthodes
- Par convention le nom d'une classe commence par une majuscule

Représentation schématique d'une classe en UML

PointPlan
-double abscisse -double ordonnée
+PointPlan(double x, double y) +translate(double dx, double dy)

Exemple de définition d'une classe en Java

```
// Fichier PointPlan.java
// Définition de la classe PointPlan

public class PointPlan
{
     // Variables d'instances –ie attributs

     private double abscisse ;
     private double ordonnée ;

     // Constructeur : Initialise les valeurs des attributs lorsqu'une
     // nouvelle instance (objet) est créé
     // Ici, les coordonnées du nouveau point sont initialisées avec
     // les deux valeurs x et y passées en paramètre

     public PointPlan( double x, double y) {
          this.abscisse = x ;
          this.ordonnée = y ;
     }

     // Méthode décrivant le comportement de l'objet
     // translate permet de « déplacer un point »

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

Déclaration et instanciation et des Objets

Les objets :

- *des données* qui définissent l'objet et décrivent son état.
- *des méthodes* qui définissent les comportements associés à l'objet (actions que peut avoir l'objet d'autres objets, ou actions qui peuvent être appliquées sur l'objet lui-même-*ie* sur son état)

Exemple d'instanciation d'un objet

```
// Fichier TestPointPlan.java
// Définition d'un programme principal permettant de tester
// la classe PointPlan

public class TestPointPlan
{
     // programme principal

     public static void main( String [] args )
     {
          // déclaration d'une variable référence
          // p pour un objet de type PointPlan

          PointPlan p ;

          // Instanciation d'un nouvel objet PointPlan et
          // affectation de sa référence à la variable p

          p = new PointPlan( 4., 5.2) ;
     }
}
```

La déclaration d'une variable

La déclaration consiste à réserver en mémoire l'espace nécessaire au stockage de la référence d'une instance : Par défaut la référence est ***null*** tant qu'aucune affectation n'a eu lieu.

Schéma :

Instanciation et référence sur les Objets

Le mécanisme d'instanciation

L'instanciation d'un nouvel objet est réalisée par l'appel à l'opérateur **`new`** et au constructeur de la classe ici **`PointPlan`**.

- **`new`** crée un objet de la classe correspondant au constructeur qui le suit
- le constructeur initialise l'objet, ici l'appel au constructeur
  `PointPlan( 4.f, 5.2f)`
  initialise un nouveau point avec une abscisse de 4 et une ordonnée de 5.2.
- **`new`** renvoie une référence vers l'instance de **`PointPlan`**
- ici la référence est affectée à la variable **`p`**

Schéma :

Remarque :

- La déclaration d'une variable réserve une zone mémoire et l'associe au nom de variable
- Le contenu associé au nom de variable dépend du type déclaré
  - **Type primitif** : on stocke une valeur
  - **Objet** : on stocke une référence vers l'objet

Variable d'instance et mémoire

Chaque instance/objet possède ses propres variables d'instance et donc son propre espace mémoire :

```
{
      PointPlan p1 ;
      PointPlan p2 ;

      p1 = new PointPlan( 0.f, -7.3f) ;
      p2 = new PointPlan( 4.f, 5.2f) ;
}
```

Schéma :

Variable-Référence

Référence et gestion de la mémoire

Les principes de la programmation objet confie aux méthodes d'instance la charge de modifier ou d'interroger l'état (les valeurs) de l'instance.

Chaque instance/objet possède ses propres variables d'instance et donc son propre espace mémoire. On accède à cet espace par une variable référence (ici **p**).

```
{
      PointPlan p = new PointPlan( 0.f, -7,3f);

      p.translate( 4.f, 4.f) ;
}
```

Schéma :

Maintient d'une référence et Garbage collector

Si une instance n'est plus référencée on ne peut plus y accéder. Toute instance non référencée est éliminée par le ramasse-miette (Garbage collector) et libère la mémoire précédemment réservée au stockage de l'objet.

```
{
      PointPlan p = new PointPlan( 0.f, -7,3f);

      P = null ;

      p.translater( 4.f, 4.f) ;     // Erreur à l'exécution
}
```

Schéma :

Variable-Référence

Maintient d'une référence et multiplicité des références

Plusieurs variables-référence peuvent référencer la même instance.

```
{
      PointPlan p1 = new PointPlan( 0.f, -7,3f);
      PointPlan p2 = p1 ;

      P1 = null ;

      P2.translate( 4.f, 4.f) ;       // Pas d'erreur à l'exécution
}
```

Schéma :

Tant qu'il existe une variable ou une structure (un tableau par ex.) dans laquelle est stockée la référence d'un objet donné, celui-ci n'est pas éliminé par le garbage collector.

Structure d'une classe 1/3

```
public class NomDeLaClasse
{
     // ------------------------------------------------
     // Déclaration des              VARIABLES D'INSTANCE
     // ------------------------------------------------
     //    elles définissent l'objet (ie description
     //    des données qui composent les instances)

     // ------------------------------------------------
     // Définition des                        CONSTRUCTEURS
     // ------------------------------------------------
     //    ils définissent comment seront initialisées
     //    les variables d'instance

     // ------------------------------------------------
     // Définition des                              MÉTHODES
     // ------------------------------------------------
     //    elles définissent le comportement des instances
}
```

Variable d'instance / attributs

La syntaxe de déclaration d'une variable d'instance :

- *Un modificateur d'accès* (**public, private, final, static** …).
- *Un type* (primitif, ou nom de classe)
- *Un nom de variable*

```
// Fichier PointPlan.java
// Définition de la classe PointPlan

public class PointPlan
{
     // Variable d'instance –ie attributs

     private double abscisse ;
     private double ordonnée ;

     private String nom;
}
```

Par convention les noms de variable commencent par une minuscule. Ces noms *DOIVENT* être expressifs.

Structure d'une classe 2/3

Constructeurs

La fonction d'un constructeur est d'initialiser les variables d'instance lors de la création d'un nouvel objet :

- Méthode particulière invoquée par l'opérateur **`new`**
- *Le nom du constructeur est le même que le nom de la classe*,
- Sauf très rares exceptions c'est toujours une méthode **`public`**.
- Elle ne renvoie pas de valeur (le type de retour n'est pas donné, même pas **`void`**)
- Il peut y avoir plusieurs constructeurs (ie. Plusieurs méthodes)
    - Elles portent toutes le même nom
    - Elles ont un nombre d'argument et des types d'arguments qui sont différents

```
public class PointPlan
{
     // Variable d'instance –ie attributs
     private double abscisse ;
     private double ordonnée ;

     // Constructeur Champ à Champ : Initialise un nouveau
     // point avec les coordonnées passées en paramètre
     public PointPlan( double x, double y) {
          this.abscisse = x ;
          this.ordonnée = y ;
     }

     // Constructeur par défaut : Initialise un nouveau point
     // avec des coordonnées par défaut des coordonnées qui le
     // place à l'origine du référentiel
     public PointPlan() {
          this.abscisse = 0. ;
          this.ordonnée = 0. ;
     }
}
```

Exemple d'appel

```
public class TestPointPlan
{
     public static void main( String [] args )
          PointPlan p1 = new PointPlan( 0., -7,3);
          PointPlan p2 = new PointPlan( );
     }
}
```

Schéma :

Structure d'une classe 3/3

Méthode d'instance

Les méthodes d'instance sont les méthodes (fonctions) qui peuvent s'appliquer sur chaque instance (chaque objet) de la classe. Les méthodes sont donc :

- définies dans la classe,
- appliquées/utilisées sur des instances de cette classe (des objets).

Définition des méthodes d'instance

```
public class PointPlan
{
     // Variable d'instance –ie attributs
     private double abscisse ;
     private double ordonnée ;


     // Méthode décrivant le comportement de l'objet
     // Translation du point
     public void translate( double dx, double dy)
     {
          this.abscisse += dx ;
          this.ordonnée += dy ;
     }


     // Méthode décrivant le comportement de l'objet
     // Distance entre le point et l'origine du repère
     public double distanceAOrigine()
     {
          double dist ;
          dist = this.abscisse * this.abscisse + this.ordonnée *
this.ordonnée ;
          return Math.sqrt( dist) ;
     }
}
```

Remarque :

- **dx** et **dy** sont des paramètres de la méthode **translate()**
- Les paramètres peuvent être de type atomique ou des références vers des instances
- Les paramètres sont typés et nommés

Remarque :

- La méthode **distanceAOrigine**() renvoie une valeur numérique de type **double**, alors que la méthode **translate()** ne renvoie rien (**void**).

Remarque :

- **double dist ;** est une variable locale à la méthode **distanceAOrigine()**

Appel des méthodes

Syntaxe

Pour appeler une méthode sur un objet on utilise la syntaxe de la notation pointée où :

- *le préfixe* est le nom de la variable référence de l'instance,
- *le suffixe* est constitué du nom de la méthode et de ses paramètres entre parenthèses.

```
nomVariableReference.nomMéthode( params,  …)
```

Exemple d'appel

```
public class TestPointPlan
{
     public static void main( String [] args )

     // Déclaration et instanciation de 2 PointPlan

          PointPlan p1 = new PointPlan( 0., -7,3);
          PointPlan p2 = new PointPlan( );

     // Appel de la méthode translate sur chaque instance

          p1.translate( 1., 3.)
          p2.translate( -1., 7.);
     }
}
```

Schéma :

Passage des paramètres

Les valeurs des paramètres des méthodes peuvent être de 2 types : soit une référence vers l'instance d'une classe, soit une valeur d'un type atomique.

Si le paramètre de la méthode est

- **l'instance o d'une classe,** *alors ce qui est transmit est* **une référence vers l'instance,**
- **une valeur d'un types atomiques,** *alors ce qui est transmit est* **une copie de la valeur.**

this : une référence sur l'instance courante

Dans le corps des instructions d'une méthode $\mathcal{M}$, il faut pouvoir faire référence à l 'instance courante (celle sur laquelle est utilisée la méthode $\mathcal{M}$).

Le mot clef **`this`**, contient une référence vers l'instance d'appel de la méthode.

```
public class PointPlan
{
     private double abscisse ;
     private double ordonnée ;

     public void translateX( double abscisse )
     {
          this.abscisse = this.abscisse + abscisse ;
     }
}
```

```
public class TestPointPlan
{
     public static void main(String [] args)
     {
          PointPlan p1 = new PointPlan() ;
          PointPlan p2 = new PointPlan(4., 5.2) ;

          p1.translateX ( 7.) ;
          p2.translateX ( 9.) ;
     }
}
```

Schéma :

this() : une référence à un constructeur déjà défini

Dans le corps des instructions des méthodes d'une classe A, il faut pouvoir faire parfois référence à un constructeur de cette même classe. Cela est possible par l'appel à la méthode **`this(…)`**. Une bonne façon de procéder est de définir le constructeur le plus général possible puis d'utiliser la/les méthodes **`this()`** pour définir les autres constructeurs.

Attention, `this()` ne peut être que la première instruction du nouveau constructeur.

```
public class PointPlan
{
     private double abscisse ;
     private double ordonnée ;

     // Constructeur Champ à Champ (le plus général)
     // défini en utilisant  un style de programmation impérative
     // utilisation explicite du symbole d'affectation (=)
     public PointPlan( double x, double y) {
          this.abscisse = x ;
          this.ordonnée = y ;
     }

     // Constructeur par défaut (plus spécifique)
     // fait appel au constructeur champ à champ
     public PointPlan() {
          this( 0., 0.) ;
     }
}
```

```
public class TestPointPlan
{
     public static void main(String [] args){
          // appel au constructeur Champ à Champ
          PointPlan p2 = new PointPlan(4., 5.2) ;
          // appel au constructeur par défaut
          PointPlan p1 = new PointPlan() ;
     }
}
```

Schéma :

L'encapsulation

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

- Le mot clef **`private`** indique que la donnée ou la méthode est inaccessible (et invisible) depuis l'extérieur de la classe.
- Le mot clef **`public`** signale qu'il n'y aura aucune restriction d'accès.

Le principe d'encapsulation est un principe *fort et structurant* de la programmation objet. Il faut le respecter. Il consiste :

- **Simplifier la tâche de l'utilisateur** en lui masquant les détails internes qui ne le concernent pas en tant qu'utilisateur de la classe,
- **Protéger le contenu des instances** en rendant impossible l'accès direct aux variables (**`private`**) ou aux méthodes qui n'appartiennent pas à la *vue publique*.
- **Permet de rendre l'utilisation d'une classe indépendante de son implémentation** interne. Ainsi, la partie privée peut changer (modification, optimisation, adaptation, …) sans que l'utilisateur n'ai à faire de modifications dans son code, pour peu que la partie publique ne soit pas modifiée.

```
public class PointPlan
{
     // Variable d'instance –ie attributs
     private double abscisse ;
     private double ordonnée ;

     public String nom ;    // Très déconseillé

     public PointPlan( double x, double y) {
          this.abscisse = x ;
          this.ordonnée = y ;
     }

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

     private double distAOrigine()
     {
          double dist ;
          dist = this.abscisse * this.abscisse + this.ordonnée *
this.ordonnée ;
          return Math.sqrt( dist) ;
     }

}
```

L'encapsulation - 2

Vue privée (interne)

PointPlan
- double abscisse - double ordonnée + String nom
+ PointPlan(…) + void translate(…) - double distAOrigine()

Vue publique (externe)

PointPlan
+ String nom + PointPlan(…)
+ Void translate(…)

Pour les attributs

- **private** défini un attribut qui n'est accessible (lecture et écriture) que par une instruction du corps d'une méthode de la classe
- **public** défini un attribut qui est accessible (lecture et écriture) par une instruction qui peut être écrite dans toute méthode y compris dans une méthode d'une autre classe.

Pour les méthodes

- **private** définit une méthode qui ne peut être invoquée que dans le corps d'une méthode de la classe où elle est définie
- **public** définit une méthode qui peut être invoquée dans le corps de n'importe quelle méthode de n'importe quelle classe

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

          p.abscisse = 4 ;          // provoque un rejet du compilateur
                                    // abscisse est inconnue car private

          p.nom = « M » ;           // OK : nom est connu et accessible
                                    // (car déclaré comme public

          double d = p.distAOrigine() ;      // rejet du compilateur

          p.translate( 3., 10.) ;// OK
     }
}
```

L'encapsulation - Technique

- 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 (**getter**)
  - En écriture : définir une **méthode publique** affectant à la variable la valeur passée en paramètre de la méthode (**setter**).
  - TESTER en plaçant la méthode **`main()`** dans une classe de test différente de la classe de définition.

```
public class PointPlan
{
     // Les attributs sont privés
     private double abscisse ;
     private double ordonnée ;

     // Méthode publique d'accès en lecture
     public double getAbscisse() {
          return this.abscisse ;
     }

     // Méthode publique d'accès en écriture
     public void setAbscisse(double x) {
          this.abscisse = x ;
     }
}
```

La classe de test doit être séparée puisque dans la classe **`PointPlan`**, même les attributs privés sont accessibles. Donc si l'on veut vérifier la limitation de portée il faut le faire depuis une classe différente !

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

          // instructions valides
          double x = p.getAbscisse() ;
          p.setAbscisse( 3.9) ;

          // rejet du compilateur
          double x = p.abscisse ;
          p. abscisse= 3.9 ;
     }
}
```

L'encapsulation - Exemple

On limite le style de programmation procédurale pour favoriser la programmation objet.

```
public class PointPlan
{
     // ------------------------------------------------------------
     // Les attributs sont privés

     private double abscisse ;
     private double ordonnée ;

     // ------------------------------------------------------------
     // Les constructeurs sont publics et font appel au accesseurs

     public PointPlan( double x, double y) {
          this.setAbscisse(x) ;
          this.setOrdonnée(y) ;
     }

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

     // ------------------------------------------------------------
     // Les accesseurs ne sont publics que si ils font partie de la
     // vue publique de la classe

     public double getAbscisse() {
          return this.abscisse ;
     }

     public void setAbscisse(double x) {
          this.abscisse = x ;
     }

     // ------------------------------------------------------------
     // Les autres méthodes ne sont publiques que si elles font
     // partie de la vue publique de la classe et elles utilisent
     // à chaque fois que c'est possible les méthodes déjà définies

     public void translate( double dx, double dy){
          this.setAbscisse( this.getAbscisse() + dx) ;
          this.setOrdonnée( this.getOrdonnée() + dy) ;
     }

     private double distAOrigine(){
          return Math.sqrt(   Math.pow(this.getAbscisse(), 2) +
                              Math.pow(this.getOrdonnée (), 2)  );
     }

}
```

Javadoc

Un code non commenté n'est pas un code de qualité. Il a une durée de vie de quelques jours et uniquement dans l'esprit de son concepteur.

Parmi les bonnes pratiques, nécessaires en Java, figure la documentation du code. Celle-ci passe par l'ajout de commentaires. Pour la Javadoc, il existe un format spécifique qui est reconnu par certains programmes et qui permet de générer automatiquement une documentation au format HTML. Ce format s'appuie sur des mots clefs/tag :

Tag	Rôle
@author	Donne le ou les auteurs de l'élément
@exception	Décrit si une exception peut être levée par la classe ou la méthode
@param	Décrit un paramètre d'une méthode. Il faut autant de tag qu'il y a de paramètres.
@return	Décrit la valeur renvoyée. Ne rien mettre si la méthode renvoie **`void`**.
@version	Donne le numéro de version de l'élément

Voir dans les pages de man : `man javadoc`
Les commentaires sont encadrés par une syntaxe *ad hoc*. Notamment, dans le code suivant, la double astérisque de la première ligne est obligatoire.

```
/**
 * Description
 *
 * @tag1
 * @tag2
 */
```

La génération d'une documentation au format HTML passe par la commande :

```
login@bash>
login@bash> mkdir doc/
login@bash> javadoc -encoding utf8  -charset utf8 -d
doc/ PointPlan.java
```

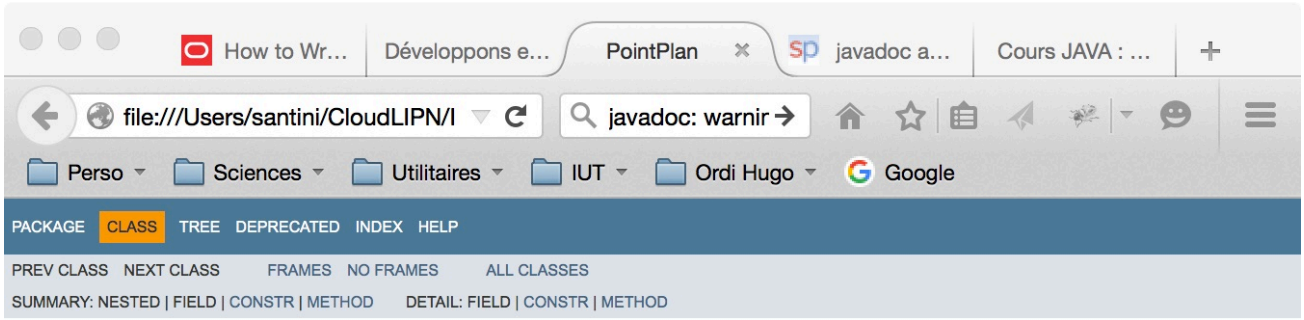
Javadoc

Exemple de commentaires/documentation

```
/**
 * Décrit un point du plan 2D
 * @version 1.0
 * @author Guillaume Santini
 */
public class PointPlan
{
        // ------------------------------------------------
        // Variables d'instance
        private double abscisse ;
        private double ordonnée ;
        private String nom ;

        // ------------------------------------------------
        // Constructeurs

        /**
         * Constructeur Champ à Champ : Initialise un nouveau
         * point avec les coordonnées passées en paramètre
         * @param x abscisse du point
         * @param y ordonnée du point
         * @param nom nom du point
         */
        public PointPlan( double x, double y, String nom)
        {
                this.setAbscisse( x ) ;
                this.setOrdonnée( y ) ;
                this.setNom( nom ) ;
        }

        /**
         * Constructeur par défaut : Initialise un nouveau point
         * avec des coordonnées par défaut des coordonnées qui le
         * place à l'origine du référentiel
         */
        public PointPlan()
        {
                this( 0., 0., "M") ;
        }
        . . .
        . . .
        . . .
        . . .
}
```

Javadoc

Donne après compilation de la documentation un fichier HTML de la forme suivante :



Class PointPlan

java.lang.Object
PointPlan

```
public class PointPlan
extends java.lang.Object
```

Décrit un point du plan 2D

Constructor Summary

Constructors

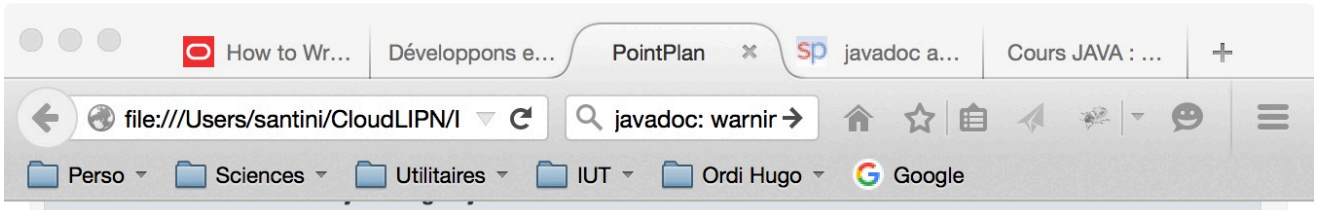
Constructor and Description
PointPlan() Constructeur par défaut : Initialise un nouveau point avec des coordonnées par défaut des coordonnées qui le place à l'origine du référentiel
PointPlan(float x, float y) Cosntructeur permettant d'initialiser un point sans avoir à lui donner un nom.
PointPlan(float x, float y, java.lang.String nom) Constructeur Champ à Champ : Initialise un nouveau point avec les coordonnées passées en paramètre

Method Summary

All Methods | **Instance Methods** | **Concrete Methods**

Modifier and Type	Method and Description
PointPolaire	**coorPolaires()** Les coordonnées polaires du point
double	**distanceAOrigine()** Distance entre le point et l'origine du repère
float	**getAbscisse()** Méthode publique d'accès en lecture a abscisse
java.lang.String	**getNom()** Méthode publique d'accès en lecture au nom du point
float	**getOrdonnée()** Méthode publique d'accès en lecture a l'ordonnée
void	**setAbscisse(float x)** Méthode publique d'accès en écriture a l'abscisse
void	**setAbscisse(java.lang.String nom)** Méthode publique d'accès en écriture au nom

Javadoc




```
clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait
```

Constructor Detail

PointPlan

```
public PointPlan(float x,
                 float y,
                 java.lang.String nom)
```

Constructeur Champ à Champ : Initialise un nouveau point avec les coordonnées passées en paramètre

Parameters:

`x - abscisse du point`

`y - ordonnée du point`

`nom - nom du point`

PointPlan

```
public PointPlan()
```

Constructeur par défaut : Initialise un nouveau point avec des coordonnées par défaut des coordonnées qui le place à l'origine du référentiel

PointPlan

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

Cosntructeur permettant d'initialiser un point sans avoir à lui donner un nom. Le nom est "M" par défaut

Parameters:

`x - abscisse du point`

`y - ordonnée du point`

Method Detail

getAbscisse

```
public float getAbscisse()
```

Méthode publique d'accès en lecture a abscisse

Returns:

`la valeur de l'abscisse`

setAbscisse

```
public void setAbscisse(float x)
```

Méthode publique d'accès en écriture a l'abscisse

Exercices #1

Soit la classe suivante :

```
// définition d'une classe pour un vaisseau
public class Vaisseau {
     private int nbMaxPassagers;   // Equipage + passager
     private String catégorie;     // Classe du vaisseau
     public double altitude;       // altitude effective de vol

     // Constructeurs -------------------------------------
     /**
      * Constructeur champ à champ
      */
     public Vaisseau( String cat, int nbPass, double alt) {
          this.setNbMaxPassagers( nbPass) ;
          this.setCatégorie( cat) ;
          this.altitude = alt ;
     }

     // Accesseur Getter/Setter  -------------------------
     /**
      * Fixe la catégorie du vaisseau
      * @param cat nom de la catégorie
      */
     public void setCatégorie( String cat){
          this.catégorie = cat;
     }

     /**
      * Retourne la catégorie du vaisseau
      * @return la catégorie du vaisseau
     */
     public String getCatégorie(){
          return this.catégorie;
     }

     /**
      * Fixe la capacité en nombre de passagers du vaisseau
      * @param n nombre maximal de passagers
      */
     public void setNbMaxPassagers( int n){
          this. nbMaxPassagers = n;
     }

     /**
      * Retourne la capacité en nombre de passagers du vaisseau
      * @return nombre maximal de passagers
     */
     public int getNbMaxPassagers(){
          return this.nbMaxPassagers;
     }
}
```

Question 1 : Identifiez et donnez la liste des variables d'instance, des constructeurs et des méthodes d'accès en lecture (getter) et en écriture (setter) de cette classe.

Question 2 : Donnez la vue publique et la vue privée de cette classe ?

Question 3 : Ajoutez le constructeur par défaut qui initialise un vaisseau comme étant de la catégorie « Vaisseau Léger », admettant un maximum de 7 personnes à son bord et positionné au sol (altitude 0).

On ajoute la méthode suivante à la classe **`Vaisseau`** :

```
1    public static void main(String [] args)
2    {
3         Vaisseau xwingT65 ;
4         xwingT65 = new Vaisseau("Chasseur Léger", 2, 10000) ;
5         Vaisseau stalker;
6         stalker = new Vaisseau("Vaisseau Lourd", 46785, 1800000) ;
7         Vaisseau surprise = xwingT65 ;
8
9         System.out.println(stalker.getAltitude()) ;
10        System.out.println(stalker.nbMaxPassagers) ;
11        System.out.println(surprise.getAltitude ()) ;
12
13        xwingT65.setAltitude(10) ;
14        xwingT65.nbMaxPassagers = 1 ;
15        System.out.println(xwingT65.getNbMaxPassagers()) ;
16        xwingT65= stalker ;
17   }
```

Question 4 : Dessinez les classes et les instances des instructions des lignes 3 à 7.

Question 5 : Quels sont les affichages produits par ce programme ?

Question 6 : Redessinez les instances à la fin de l'exécution du programme.

Question 7 : Cette classe respecte-t-elle le principe d'encapsulation ? Si ce n'est pas le cas proposez les modifications nécessaires.

Placez la méthode **`main`** précédemment définie dans une classe **`TestVaisseau`** comme présenté ci-après:

```
1 // définition d'une classe de Test pour un vaisseau
2 public class TestVaisseau
3 {
4    public static void main(String [] args)
5    {
6          [...]

20    }
21 }
```

Question 8 : Identifiez les instructions rejetées par le compilateur, expliquez pourquoi certaines d'entre elles fonctionnaient lorsque la méthode **`main`** était dans la classe **`Vaisseau`**.

Question 9 : Pourquoi est-il important de confier le test d'une classe à une autre classe (une classe de test dédiée) ?

Question 10 : Proposez une adaptation valide du code de la méthode **`main`** de la classe **`TestVaisseau`**.

Exercices complémentaires

Question 11 : Définissez une classe **`Droid`** qui sera caractérisée par un modèle et une taille. Cette classe proposera un constructeur par défaut et un constructeur champ à champ et respectera les règles d'encapsulation.

Question 12 : Définissez une classe **`TestDroid`** proposant un ensemble d'instructions permettant de vérifier l'encapsulation de tester la vue publique de la classe. Vous pourrez par exemple instancier un de modèle « série R2 » de 96cm de haut et un autre de modèle « 3PO »