Spécification et implantation
Corrigé

Même s'il est habituellement recommandé de lire tout l'énoncé avant d'y répondre, nous vous demandons exceptionnellement ici de de pas anticiper la lecture des questions.

Exercice 1 : Préambule : qu'est ce qu'un point géométrique ?

1.1. Recenser tout ce qui est défini sur le point géométrique.

Solution : Pas de solution type pour cette question. C'est à chacun de jouer le jeu et de proposer quelque chose...

1.2. Indiquer les éléments recensés en 1.1 qui peuvent être représentés sous la forme d'attribut.

Solution : Souvent, on raisonne surtout en terme d'état et donc d'attributs.

S'il n'y a que des attributs, c'est que les opérations/méthodes ont été oubliées !

1.3. Quel formalisme a été utilisé pour répondre à la question 1.1 ?

Solution : Simplement écrire en vrac, ou en colonne, n'est pas suffisant. Il faut utiliser UML et plus particulièrement la représentation d'une classe en UML. L'intérêt principal c'est qu'UML fait apparaître explicitement les rubriques « attributs » et « opérations ». On a donc un pense-bête, on n'oubliera pas d'identifier les opérations.

Raisonner en terme d'attributs et d'opérations est un point de vue de programmeur (comment on écrit la classe) et donc *implantation*. Dans un première temps, il faut répondre à la question « quoi ? » : que doit fournir la classe ? C'est le point de vue des utilisateurs de la classe qu'il faut d'abord adopter. On doit *spécifier* ce qu'ils trouveront sur la classe. C'est ce que visent à (dé)montrer les exercices suivants.

Exercice 2 : Spécifier en programmation objet : exemple du point géométrique

Quand on veut spécifier un type en programmation objet, on doit prendre le point de vue des utilisateurs de ce type. On ne s'intéresse donc qu'aux méthodes auxquelles ils auront accès. Ces méthodes sont classées en deux catégories :

— les *requêtes* : informations qui peuvent être demandées à un objet instance de la classe (méthodes sans effet de bord, i.e. fonctions) ;

— les *commandes* : services qui peuvent être réalisés par un objet (équivalent de procédures).

Spécifier le point géométrique décrit ci-après, c'est-à-dire identifier les requêtes et commandes, dessiner le diagramme d'analyse [1] UML correspondant, ajouter les informations de type, préciser la sémantique (objectif, conditions d'utilisation et effets) des requêtes et commandes.

1. On appelle diagramme d'analyse UML le diagramme de classes où les rubriques *attributs* et *opérations* sont remplacées par *requêtes* et *commandes*. Attention, *requêtes* ne correspond pas à *attributs*, ni *commandes* à *opérations* : *requêtes* et *commandes* correspondent à *opérations*.

On s'intéresse aux points du plan en considérant aussi bien leurs coordonnées cartésiennes (abscisse et ordonnée) que polaires (module et argument). On souhaite pouvoir obtenir la distance qui sépare deux points et translater un point en précisant un déplacement suivant l'axe des X (abscisses) et un déplacement suivant l'axe des Y (ordonnées). On veut pouvoir afficher un point sous la forme du couple (abscisse,ordonnée). Enfin, on veut pouvoir modifier l'abscisse, l'ordonnée, le module ou l'argument d'un point.

Solution :

Les requêtes sont : x, y, module, argument, distance.

Les commandes sont : afficher, translater, setX, setY, setModule, setArgument

Le diagramme UML correspondant est alors :

Point
requêtes
x : double
y : double
module : double
argument : double
distance(autre : Point) : double
commandes
translater(dx : double, dy : double)
afficher()
setX(nx : double)
setY(ny : double)
setModule(module : double)
setArgument(argument : double)

Remarque : Nous avons fait de `afficher()` une commande car elle ne retourne rien et se contente d'écrire sur le terminal. Si nous avions suivi la convention Java, nous aurions pu définir une méthode `toString()` qui retourne une chaîne de caractères correspondant à la représentation de ce point. Dans ce cas, on aurait eu une *requête*.

Remarque : Pour setX et setY, nous avons appelé nx et ny le paramètre (nouveau x et nouveau y). On aurait pu les appeler x et y comme les requêtes (choix qui a été fait pour setModule et setArgument).

En dessinant le diagramme d'analyse ci-dessus, nous avons juste identifié les requêtes et commandes et donné leur signature. Il faudra aussi donner leur documentation et la formaliser en utilisant la programmation par contrat (ceci peut apparaître sur le diagramme d'analyse sous la forme d'annotations).

Ainsi, on peut identifier les invariants suivants qui lient coordonnées cartésiennes et coordonnées polaires et qui expriment que le module est toujours positif :

```
1 x == module * Math.cos(argument);
2 y == module * Math.sin(argument);
3 module >= 0
```

On pourrait aussi normaliser l'argument (par exemple compris dans $[0, 2\pi[$). Ceci se traduirait pas deux invariants supplémentaires :

```
argument >= 0;
argument < 2 * Math.PI;
```

On peut formaliser ainsi les conditions d'utilisation et les effets des requêtes et des commandes :

— une précondition de distance est que l'autre point existe :

```
@requires autre != null;
@ensures \result == Math.sqrt(Math.pow(this.x - autre.x, 2),
                     Math.pow(this.y - autre.y, 2));
```

— l'effet de translater et d'augmenter x et y de dx et dy respectivement :

```
@ensures this.x == \old(this.x) + dx;
@ensures this.y == \old(this.y) + dy;
```

— l'effet de setX(nx : int) est de changer la valeur de X en laissant Y inchangé.

```
@ensures this.x == nx;
@ensures this.y == \old(this.y); // Il existe une autre manière en JML de le dire
```

— l'effet de setModule(nm : int) est de changer le module, l'argument restant inchangé (dans le cas où nm vaut 0, il serait logique de mettre l'argument à 0 aussi !). Il est nécessaire que le module soit positif.

```
@requires nm >= 0;
@ensures this.module == nm;
@ensures nm != 0 ==> this.argument == \old(this.argument);
@ensures nm == 0 ==> this.argument == 0;
```

Attention : Nous avons utilisé l'égalité entre réels. En réalité, il faudrait vérifier que les réels sont proches à *epsilon* près...

Exercice 3 : Définir les constructeurs

3.1. Qu'est-ce qu'un constructeur ?

Solution : Un constructeur est quelque chose qui ressemble à une méthode et qui permet d'initialiser une instance d'une classe et plus généralement de réaliser ce que l'on souhaite à la création d'un objet.

Une classe peut-elle avoir plusieurs constructeurs ?

Solution : Une classe peut avoir autant de constructeurs qu'on le souhaite... dans la limite de la surcharge ! La seule contrainte est en effet de trouver une signature différente pour chaque constructeur.

Nous verrons dans la suite que cette limite peut arriver très vite !

Quel est l'intérêt d'un constructeur ?

Solution : Le compilateur garantit que pour chaque création d'un objet, un constructeur lui est appliqué garantissant qu'il est initialisé comme prévu par l'auteur de la classe de l'objet. Le compilateur peut signaler les initialisations oubliées (pas de constructeur applicable).

Un constructeur garantit que l'objet est initialisé à partir des informations fournies par l'utilisateur. Le constructeur permet de rendre atomique la réservation de l'espace mémoire et son initialisation.

3.2. Spécifier un constructeur qui initialise un point à partir de ses coordonnées cartésiennes.
Solution :

Point
requêtes
x : double
y : double
module : double
argument : double
distance(autre : Point) : double
commandes
translater(dx : double, dy : double)
afficher()
setX(nx : double)
setY(ny : double)
setModule(module : double)
setArgument(argument : double)
constructeurs
Point(vx : double, vy : double)

3.3. Peut-on spécifier un second constructeur pour initialiser un point à partir de ses coordonnées polaires ? Pourquoi ?
Solution : Ce nouveau constructeur serait :

```
Point(module: double, argument: double)
```

On a donc la même signature que le premier. Ceci est interdit !

Notons que ces deux constructeurs peuvent être définis en UML. Il suffit de leur donner deux noms différents. C'est le langage Java qui ne le permet pas.

En Java, on peut contourner ce problème (voir exercice 7).

3.4. Après avoir défini les constructeurs précédents, a-t-on accès au constructeur par défaut (celui qui ne prend pas de paramètres) ?
Solution : Non, nous n'avons plus accès au constructeur par défaut. Seuls existent les constructeurs explicitement définis et nous n'avons pas défini le constructeur sans paramètres.

A-t-on intérêt à le définir ? Pourquoi ?
Solution : Non, il n'est pas souhaitable de le déclarer car alors, le compilateur ne peut plus détecter si une initialisation a été oubliée. Le but n'est pas seulement que l'état de l'objet soit cohérent mais surtout qu'il soit initialisé avec les bonnes informations, celles que doit fournir l'utilisateur.

Ce problème sera plus probant avec l'héritage.

Exercice 4 : Implanter en programmation objet

Maintenant que la spécification des points est terminée, on peut s'intéresser aux implantations possibles. Il s'agit de définir pour une classe :

— son état représenté par les *attributs* ;

— son comportement décrit par ses *méthodes*.

Les commandes et les requêtes deviennent des méthodes. Une requête peut correspondre à une information stockée (si un attribut lui correspond) ou calculée à partir de l'état de l'objet.

4.1. *Choix d'implantation.*

4.1.1. Proposer plusieurs implantations de la spécification d'un point géométrique (en utilisant un diagramme de classe attributs/méthodes) et indiquer dans quelles conditions les utiliser.

Solution : Voici quelques implantations possibles :

— On stocke les coordonnées cartésiennes (2 attributs).

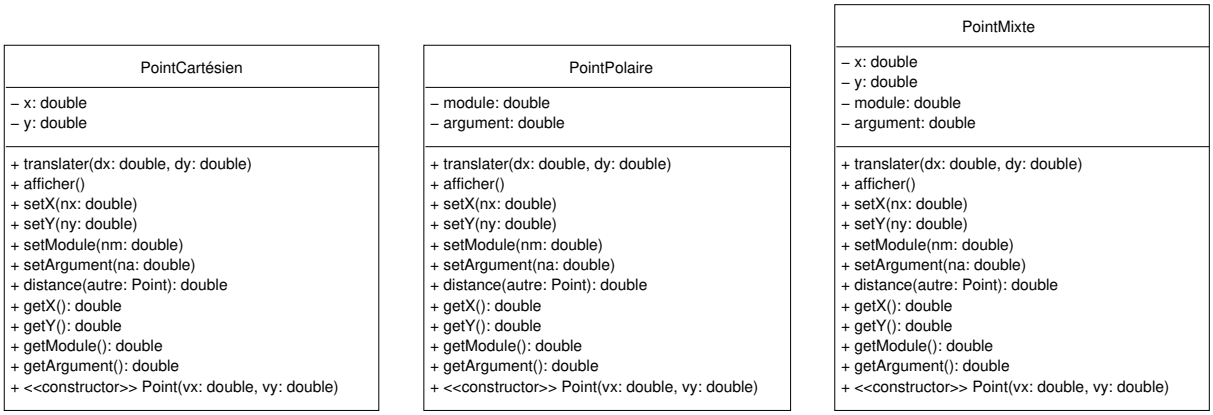
— On stocke les coordonnées polaires (2 attributs).

— On stocke les coordonnées cartésiennes et les coordonnées polaires (4 attributs). Cette implantation est par exemple utile si on modifie peu souvent les coordonnées et qu'on les consulte très souvent. On a une sorte de cache : toutes les coordonnées sont calculées lors d'une modification et sont donc immédiatement disponible lors d'une consultation (un peu de perte de classe, un peu de gain de temps).

— On stocke deux attributs réels qui représentent soit les coordonnées cartésiennes, soit les coordonnées polaires en fonction de la valeur d'un booléen qui représente le système de coordonnées (3 attributs).

— On pourrait envisager d'autres représentations...

On en déduit les trois représentations UML suivantes (en redonnant le sens attributs/méthodes aux deux rubriques). Bien noter que les méthodes sont exactement les mêmes (ce sont toutes les méthodes issues des requêtes et des commandes).



4.1.2. Écrire la méthode qui change l'abscisse du point pour chaque implantation envisagée.

Solution : Nous donnons ici le pseudo-code des différentes versiond de la méthode `setX`

1. Avec la version cartésienne : il suffit de changer la valeur de x :

```
this.x = nx;
```

2. Avec la version polaire : il faut calculer module et argument en fonction de l'ancien y et du nouvel x (attention : il faut conserver y dans une variable locale car sa valeur va changer au fur et à mesure des calculs).

```
double y = this.getY();
this.module = Math.sqrt(nx * nx + y * y);
this.argument = Math.atan2(y, nx);
```

3. Avec la version mixte : il faut calculer x, module et argument en fonction de l'ancien y et du nouvel x.

```
this.x = nx;
this.module = Math.sqrt(this.x * this.x + this.y * this.y);
this.argument = Math.atan2(this.y, this.x);
```

4. Avec la version deux attributs (a et b) + un booléen (estCartesien : a et b correspondent à x et y si vrai, à module et argument si faux) : il faut passer en coordonnées cartésienne, puis changer l'attribut a (qui correspond à x).

```
passerCartesien(); // méthode privée
this.a = nx;
```

4.2. On décide d'implanter la classe Point en choisissant l'abscisse et l'ordonnée comme seuls attributs. Écrire la classe correspondante. On se limitera aux attributs, au constructeur et aux méthodes qui permettent de translater un point, l'afficher, changer son abscisse, obtenir son module.

Solution :

La classe Point peut alors être écrite de la manière suivante.

Attention : La programmation par contrat n'a pas été ajoutée dans le texte de cette classe. Ceci aurait été possible, par exemple en utilisant JML.

```
/** Définition d'un point mathématique dans un plan qui peut être
 * considéré dans un repère cartésien ou polaire. Un point peut être affiché,
 * translaté. Sa distance par rapport à un autre point peut être obtenue.
 * @author Xavier Crégut <nom@n7.fr>
 */
public class Point {

	private double x;	// abscisse de ce point
	private double y;	// ordonnée de ce point

	/** Construire un point à partir de son abscisse et de son ordonnée.
	 * @param  vx    valeur de l'abscisse
	 * @param  vy    valeur de l'ordonnée
	 */
	public Point(double vx, double vy) {
		this.x = vx;
		this.y = vy;
	}

	/** Changer l'abscisse de ce point.
	 * @param vx la nouvelle valeur de l'abscisse
	 */
	public void setX(double vx) {
		this.x = vx;
	}

	/** Changer l'ordonnée de ce point.
	 * @param vy la nouvelle valeur de l'ordonnée
	 */
	public void setY(double vy) {
		this.y = vy;
	}

	/** Obtenir le module de ce point.
	 * @return module de ce point
	 */
```

```
37          public double getModule() {
38                return Math.sqrt(Math.pow(this.x, 2) + Math.pow(this.y, 2));
39          }
40
41          /** Obtenir l'argument de ce point.
42            * @return l'argument de ce point
43            */
44          public double getArgument() {
45                return Math.atan2(this.y, this.x);
46          }
47
48          /** Changer le module de ce point.
49            * @param nouveau_module la nouvelle valeur du module
50            */
51          public void setModule(double nouveau_module) {
52                double ancien_argument = this.getArgument();
53                this.setX(nouveau_module * Math.cos(ancien_argument));
54                this.setY(nouveau_module * Math.sin(ancien_argument));
55          }
56
57          /** Changer l'argument de ce point.
58            * @param nouvel_argument la nouvelle valeur de l'argument.
59            */
60          public void setArgument(double nouvel_argument) {
61                double m = this.getModule();
62                this.setX(m * Math.cos(nouvel_argument));
63                this.setY(m * Math.sin(nouvel_argument));
64          }
65
66          /** Afficher le point. */
67          public void afficher() {
68                System.out.print("(" + this.x + "," + this.y + ")");
69                // Remarque : il n'y a pas de retour à la ligne (print et non
70                // println) car c'est à celui qui affiche le point de savoir
71                // s'il souhaite ou non ajouter un retour à la ligne.
72          }
73
74          /** Obtenir la distance de ce point par rapport à un autre point.
75            * @param autre l'autre point
76            * @return distance avec l'autre point
77            */
78          public double distance(Point autre) {
79                double dx2 = Math.pow(autre.x - this.x, 2);
80                double dy2 = Math.pow(autre.y - this.y, 2);
81                return Math.sqrt(dx2 + dy2);
82          }
83
84          /** Translater le point de dx suivant l'axe des X et de dy suivant les Y.
85           * @param dx_ déplacement suivant l'axe des X
86           * @param dy_ déplacement suivant l'axe des Y
87           */
88          public void translater(double dx, double dy) {
89                this.x += dx;
90                this.y += dy;
91          }
92
93    }
```

Remarque : Lors de l'implantation de la classe, des méthodes non présentes dans la spécifications peuvent être ajoutées (suite à l'application de la méthode des raffinages sur les requêtes et les commandes, pour limiter la redondance de code entre les méthodes de la classe, etc.).

Ces nouvelles méthodes devraient être déclarées privées car elles n'ont pas identifiées dans la spécification.

Exercice 5 : Utiliser la classe Point

Dans la suite nous considérerons deux programmes qui utilisent la classe Point. Il s'agit ici de programmes simples mais la classe ainsi créée pourrait tout aussi bien s'intégrer dans une architecture de plus grande échelle utilisant intensivement les points.

5.1. *ExemplePoint1*. Écrire un programme qui crée un point de coordonnées cartésiennes (1,0), puis affiche son module et son abscisse, le translate de (-1, 1) et, enfin, l'affiche. Indiquer ce que doit afficher son exécution et dessiner l'évolution de la mémoire.

Solution :

```
/** Programme de test de la classe Point.
  * @author Xavier Crégut
  */
public class ExemplePoint1 {

      public static void main (String argv []) {
            Point p = new Point(1,0);
            System.out.println("abscisse = " + p.getX());
            System.out.println("module = " + p.getModule());
            p.translater(-1, 1);
            p.afficher();
            System.out.println();
      }
}
```

Le résultat de l'exécution donne :

```
abscisse = 1.0
module = 1.0
(0.0,1.0)
```

5.2. *ExemplePoint2*. Écrire un programme qui crée un point de coordonnées cartésiennes (1,0), met son abscisse à 10 et affiche son module. Indiquer ce que doit afficher son exécution.

Solution :

```
/** Programme de test de la classe Point
  * @author Xavier Crégut
  */
public class ExemplePoint2 {

      public static void main (String argv []) {
            Point p = new Point(1,0);
            p.setX(10);
            System.out.println("module = " + p.getModule());
      }
}
```

Le résultat de l'exécution donne :

```
module = 10.0
```

Exercice 6 : Comprendre pourquoi les attributs doivent être privés

Regardons les conséquences de déclarer les attributs publics dans la classe Point.

6.1. *Utiliser les attributs.* On suppose que les attributs de la classe Point ont été déclarés publics. L'utilisateur de la classe Point peut donc les utiliser directement. Indiquer les modifications qu'il

pourrait faire aux classes ExemplePoint1 et ExemplePoint2 en utilisant les attributs partout où c'est possible.

Solution : Dans ExemplePoint1, on peut directement utiliser p.x au lieu de p.getX().

```
/** Programme de test de la classe Point.
 * @author Xavier Crégut
 */
public class ExemplePoint1 {

      public static void main (String argv []) {
            Point p = new Point(1,0);
            System.out.println("abscisse = " + p.x);
            System.out.println("module = " + p.getModule());
            p.translater(-1, 1);
            p.afficher();
            System.out.println();
      }
}
```

Dans ExemplePoint2, on peut directement utiliser p.x au lieu de p.setX(...).

```
/** Programme de test de la classe Point
 * @author Xavier Crégut
 */
public class ExemplePoint2 {

      public static void main (String argv []) {
            Point p = new Point(1,0);
            p.x = 10;
            System.out.println("module = " + p.getModule());
      }
}
```

6.2. *Comprendre le principe de l'accès uniforme.* Le programmeur de la classe Point décide de changer son implantation des points. Il choisit comme attributs les coordonnées polaires, c'est-à-dire le module et l'argument. On dispose donc d'une deuxième version de la classe Point.

Est-ce que la classe ExemplePoint1 compile et s'exécute correctement ?

Solution : Dans la classe Point, version polaire, on a supprimé les attributs x et y et on a seulement les attributs module et argument.

Le programme ne compile plus il n'y a pas d'attribut x.

```
ExemplePoint1.java:8: error: cannot find symbol
            System.out.println("abscisse = " + p.x);
                                              ^

  symbol: variable x
  location: variable p of type Point
1 error
```

L'objectif est d'essayer de corriger ces problèmes en minimisant les modifications de ExemplePoint1.java (et des nombreux programmes qui utilisent les points).

Pour l'attribut x, il suffit (!!!) alors d'utiliser l'accesseur `getX()`. Mais attention, ceci oblige à changer tous les programmes qui utilisaient directement l'attribut x !

Il est donc préférable pour l'utilisateur de la classe Point de passer par les accesseurs (getX, getModule, etc.) plutôt que les attributs (x, module, etc.). Comment faire pour l'obliger à passer par les accesseurs ? Il suffit de mettre l'attribut en **`private`**.

Conclusion : Si les attributs sont déclarés `private`, les utilisateurs de la classe doivent alors obligatoirement passer par les accesseurs et leurs programmes sont alors moins sensibles à un changement d'implémentation de la classe Point.

6.3. *Comprendre le principe de protection en écriture des attributs.* Le programmeur décide de changer l'implantation de sa classe. Cette fois, il choisit pour attributs les quatre coordonnées. On dispose donc d'une troisième version de la classe Point.

Est-ce que la classe ExemplePoint2 compile et s'exécute correctement ?

Solution : La classe compile : l'attribut x est bien présent !

Voici le résultat de l'exécution.

```
module = 1.0
```

On constate alors qu'il ne donne pas les mêmes résultats qu'avec la version cartésienne des points. En fait, le module n'a pas été mis à jour ! ! !

Si on laisse l'utilisateur d'une classe directement accéder à l'état interne de cette classe (donc à ses attributs), alors on lui transfère également la responsabilité de maintenir la cohérence de la classe. Or, rien ne garantit que tous les utilisateurs seront suffisamment vigilants ! De toute façon, puisqu'il y a risque d'erreur, c'est donc qu'il y aura erreur !

Conclusion : Pour qu'une classe puisse garantir la cohérence de l'état de ses objets, il est nécessaire qu'elle ne donne pas accès à son état interne. Elle doit fournir une méthode pour changer cet état, cette méthode préservant la cohérence de l'objet.

Exercice 7 : Retour sur la création des points

On veut pouvoir créer un point en fournissant ses coordonnées cartésiennes ou ses coordonnées polaires. Comment faire ?

Solution : On a vu qu'en utilisant des constructeurs ce n'est pas possible. Les deux constructeurs auraient la même signature. La solution est alors de jouer sur le nom, et donc de définir une méthode plutôt qu'un constructeur. Cette méthode doit pouvoir créer le premier point. Il ne faut donc pas que ce soit une méthode d'instance mais une méthode de classe.

On appelle ces méthodes des **fabriques statiques**. Ce sont des méthodes de classe (qui n'ont donc pas à être appliquée sur un objet de la classe, déclarées avec le mot-clé **`static`** en Java) et qui ici vont retourner un nouveau Point. L'intérêt d'utiliser des méthodes est qu'on peut choisir leur nom. On peut, par exemple, nommer la première `cartesien` et la seconde `polaire`.

```
/** Obtenir un point à partir de ses données cartésiennes.
 * @param x l'abscisse du point à créer
 * @param y l'ordonnée du point à créer
 * @return le point d'abscisse x et d'ordonnée y
 */
public static Point cartesien(double x, double y) {
      return new Point(x, y);
}

/** Obtenir un point à partir de ses données polaires.
 * @param module le module du point à créer
 * @param argument l'argument du point à créer
 * @return le point avec le module et l'argument précisés
 */
```

```
15 public static Point polaire(double module, double argument) {
16       double x = module * Math.cos(argument);
17       double y = module * Math.sin(argument);
18       return Point.cartesien(x, y);
19 }
```

Voici un exemple d'utilisation des fabriques statiques.

```
1  /** Illustrer l'utilisation des fabrique.
2   * @author Xavier Crégut <Prenom.Nom@enseeiht.fr>
3   */
4  public class ExempleFabrique {
5
6        public static void main(String[] args) {
7              Point p1 = Point.cartesien(1, 4);
8              assert 1 == p1.getX();
9              assert 4 == p1.getY();
10
11             Point p2 = Point.polaire(2, Math.PI);
12             assert 2 == p2.getModule();
13             assert Math.PI == p2.getArgument();
14       }
15
16 }
```

En général, on limite l'accès aux constructeurs de la classe pour obliger les utilisateurs de cette classe à passer par les fabriques statiques qui sont plus explicites : on définit donc les constructeurs **`private`** ou plus souvent**`protected`**.

L'inconvénient des fabriques statiques, c'est qu'elles ne sont pas identifiées clairement dans la documentation engendrées par javadoc (elles sont au milieu des méthodes) et il est donc plus difficile de les identifier que les constructeurs qui ont une rubrique spécifique.

Objectifs :

- Faire la différence entre la spécification et l'implantation en technologie objet
- Savoir comment spécifier une classe
- Savoir comment implanter une classe
- Comprendre les notions de classe, attribut, méthode, constructeur, objet/instance, poignée, droit d'accès/visibilité
- Savoir écrire une classe en Java
- Comprendre les constructeurs
- Comprendre pourquoi les attributs doivent être privés
- Comprendre les fabriques statiques