Initiation à la Programmation Orientée Objets

Sébastien Jeudy

(Version V1)

Initiation à la Programmation Orientée Objets

Objectif :

Acquérir les bases de la Programmation Orientée Objets (POO)

Sommaire :

I. Concepts objets

1) L'objet par rapport aux autres styles de programmation
2) Classes, objets et packages
3) Méthodes et communication inter-objets
4) Agrégation et encapsulation
5) Héritage, polymorphisme, classes abstraites et interfaces

II. Introduction à la POO

1) Historique et comparaison
2) Impossibilités et enjeux : passer du procédural à l'objet
3) Syntaxe rapide généraliste de plusieurs langages accueillant l'objet

III. Utiliser l'objet

1) Les instances des objets
2) Staticité et dynamicité : correspondance avec la vie réelle

IV. Eléments principaux d'architecture

1) Les classes, attributs et méthodes : éléments fondamentaux
2) Visibilité : comment, pourquoi
3) Héritage et réutilisation du code
4) Interfaces et abstraction : préparation raisonnée d'une architecture

I. Concepts objets

I. Concepts objets

1) L'objet par rapport aux autres styles de programmation
2) Classes, objets et packages
3) Méthodes et communication inter-objets
4) Agrégation et encapsulation
5) Héritage, polymorphisme, classes abstraites et interfaces

1) L'objet par rapport aux autres styles de programmation

La programmation orientée objet (POO) a été **élaborée dans les années 1970 avec le langage Smalltalk**

C'est un paradigme de programmation informatique qui **consiste en la définition et l'interaction de briques logicielles appelées objets**

Un objet représente un concept, une idée ou **toute entité du monde physique** (comme une voiture, une personne ou encore un livre) **transposés au monde informatique**

L'orienté objet (OO) **supplante peu à peu le procédural dans les programmes importants car il présente d'énormes avantages :** facilité d'organisation, réutilisation, méthode plus intuitive, possibilité d'héritage et de polymorphisme, facilité de correction et d'évolution, projets plus faciles à gérer,...

L'intérêt principal de l'OO réside dans le fait que l'**on ne décrit plus par le code des actions à réaliser de façon linéaire, mais par des ensembles cohérents d'objets**

L'OO est **facilement concevable car il décrit des entités comme il en existe dans le monde réel :**

L'objet « Voiture », qui implémente la méthode « Voiture.Rouler() » et possède la propriété « Voiture.Cylindree », est facilement concevable et peut être utilisé – par exemple – dans une simulation de course automobiles où l'on fera interagir plusieurs objets « Voiture » différents

→ *les modèles à objets ont été créés pour modéliser le monde réel*

L'OO **permet de factoriser le code en ensembles logiques :** du point de vue programmation, l'OO permet d'écrire des programmes facilement lisibles avec un minimum d'expérience, de taille minimale et à la correction aisée. Ces programmes sont, de plus, souvent très stables

Les informations concernant un domaine étant centralisées en objets, il est **facile de sécuriser le programme en interdisant ou autorisant l'accès à ces objets** aux autres parties du programme

Deux types de développeurs cohabitent en OO : ceux qui conçoivent les objets et ceux qui utilisent ces objets dans leurs programmes

→ *la programmation objet permet de diviser la complexité*

La programmation objet **permet également d'obtenir des projets stratifiés :** chaque protagoniste ne travaille que sur des implémentations le concernant

2) Classes, objets et packages

Un objet est avant tout une structure de données : la principale différence vient du fait que l'objet regroupe les données et les moyens de traiter ces données

Un objet rassemble de fait deux éléments de la programmation procédurale :

- Les attributs (ou propriétés) sont à l'objet ce que les variables sont à un programme : ce sont eux qui ont en charge les données à gérer (tout comme n'importe quelle autre variable, un attribut peut posséder un type quelconque défini au préalable : nombre, caractère,...)

- Les méthodes sont les éléments d'un objet qui servent d'interface entre les données et le programme : sous ce nom obscur se cachent simplement des procédures ou fonctions destinées à traiter les données

→ *les attributs et les méthodes sont les membres d'un objet*

En programmation orientée objet, **une classe définit l'ensemble des attributs et des méthodes nécessaires (les membres)**

On assimilera une classe à un moule qui produit des objets : c'est un modèle pour plusieurs objets semblables, partageant des caractéristiques communes

→ *le concept de classe est abstrait, alors que le concept d'objet est fondamentalement concret*

Un objet est une instance de classe, c'est-à-dire un exemplaire utilisable créé à partir de cette classe et en valorisant certaines propriétés (une fois l'objet instancié, il correspond en réalité à une zone mémoire donnée)

Une même classe peut donc avoir plusieurs objets instanciés (normalement avec des valeurs de propriétés différentes)

Exemple :

Pierre et Paul sont des instances objets de la classe Humain, c'est-à-dire des humains ayant des propriétés spécifiques (attributs) : Pierre est un humain aux yeux bleus et Paul un humain aux yeux marrons

En tant qu'humains, Pierre et Paul peuvent tous les deux parler et marcher (méthodes), quand ils en ont la nécessité

Symbolisation (notation pointée) :

Le plus souvent, la syntaxe informatique d'un objet, faisant référence à ses attributs (propriétés) et ses méthodes, est la suivante :

Objet.NomAttribut

Objet.NomMethode(argument(s))

→ *la programmation orientée objet regroupe également d'autres concepts : encapsulation, héritage, polymorphisme,...*

Packages : → *organisation des classes (pour diffusion, réutilisation,...)*

Les classes sont organisées en packages qui forment une arborescence (cette notion est orthogonale à celle d'héritage)

→ *dépendants les uns les autres*

Un package est donc un **ensemble de classes attachées à un même concept** → *"bibliothèque logicielle"*

(par exemple, le package « java.io » contient les classes associées aux entrées/sorties, alors que le package « java.net » contient les classes spécialisées dans la programmation des réseaux)

Les packages peuvent à leur tour se ramifier en sous-packages, etc.

3) Méthodes et communication inter-objets

En programmation orientée objet (POO), **une méthode est une routine (fonction) membre d'une classe**

Une méthode peut être :

- Une méthode d'instance, n'agissant que sur un seul objet (instance de la classe) à la fois

- Une méthode statique ou méthode de classe, indépendante de toute instance de la classe (objet) → *simple fonction procédurale*

En POO, on utilise parfois le terme spécifique **"invocation de méthode" pour désigner l'appel d'une telle fonction** d'une classe

Portée des méthodes :

Dans de nombreux langages de POO, **l'encapsulation à l'intérieur d'une classe permet de gérer et donc restreindre les droits d'accès à un membre** de cette classe, soit une méthode (fonction membre) ou une propriété (donnée membre), **on parle alors de "portée du membre"**

On utilise souvent (comme en C++ ou en Java) les mots-clés "private", "protected" ou "public" pour gérer ces droits d'accès :

- Privée : accessible par les membres de la classe seulement (seul un objet de la classe peut l'appeler, couvre généralement le fonctionnement interne de la classe que l'on veut masquer de l'extérieur)

- Protégée : accessible par les membres de la classe et des classes dérivées (seul un objet de la classe ou d'une classe dérivée peut l'appeler)

- Publique : accessible par les membres de la classe et de ses dérivées ainsi que les clients de la classe (c'est-à-dire de tout objet, constitue l'interface de la classe)

Classification des méthodes : il existe différents types de méthodes

- Le(s) constructeur(s) appelé(s) à la création de l'objet (fonction particulière appelée lors de l'instanciation, permettant d'allouer la mémoire nécessaire à l'objet et d'initialiser ses attributs)

- Le destructeur appelé à la suppression de l'objet, explicitement ou bien implicitement (méthode spéciale appelée afin de récupérer les ressources, principalement la mémoire, réservées dynamiquement lors de l'instanciation)

→ *alors qu'il peut exister plusieurs constructeurs, il ne peut exister qu'un seul destructeur*

- Les accesseurs (Get) permettant de récupérer la valeur de données membres privées sans y accéder directement de l'extérieur (sécurisent donc les attributs)

- Les mutateurs (Set) permettant de modifier l'état de données membres (tout en vérifiant si la valeur que l'on veut donner à la donnée membre respecte les normes de celle-ci ou diverses règles de cohérence)

- Les méthodes abstraites qui sont des méthodes sans code (leur existence dans une classe suffit à déclarer qu'une classe est abstraite et contraint à introduire des classes filles pour les implémenter et les exploiter)

4) Agrégation et encapsulation

Composition :

Indique qu'**un objet A (appelé conteneur) est constitué d'un autre objet B**

Cet objet B n'appartient qu'à l'objet A et ne peut pas être partagé avec un autre objet

C'est une relation très forte, si l'objet A disparaît, alors l'objet B disparaît aussi

Exemple : un cheval est constitué d'une tête et de quatre jambes

Agrégation :

Indique qu'**un objet A possède un autre objet B, mais contrairement à la composition, l'objet B peut exister indépendamment de l'objet A**

La suppression de l'objet A n'entraîne pas la suppression de l'objet B

L'objet A est plutôt à la fois possesseur et utilisateur de l'objet B

Exemple : un cheval possède une selle sur son dos

Encapsulation :

C'est l'idée de **protéger l'information contenue dans un objet et de ne proposer que des méthodes de manipulation de cet objet**

L'utilisateur extérieur ne peut pas modifier directement l'information et risquer de mettre en péril les propriétés comportementales de l'objet

→ *l'objet est vu de l'extérieur comme une "boîte noire"*

Les principes de l'encapsulation sont notamment appliqués à l'aide des trois niveaux de visibilité :

- **Public :** les attributs de l'objet sont accessibles à tous
- **Protégé :** les attributs sont accessibles seulement aux classes dérivées
- **Privé :** les attributs sont accessibles seulement à l'objet lui-même

On utilise également des **méthodes d'accès et de modifications** définies dans l'une des trois catégories (suivant l'effet que l'on souhaite obtenir)

Ces **deux types de méthodes (accesseurs "Get" et mutateurs "Set")** sont alors définis en plus de l'attribut qui contient réellement la donnée

5) Héritage, polymorphisme, classes abstraites et interfaces

Héritage : mécanisme qui **permet, lors de la déclaration d'une nouvelle classe, d'y inclure les caractéristiques d'une autre classe**

→ *établit une relation de généralisation / spécialisation qui permet d'hériter dans la déclaration d'une nouvelle classe (appelée classe dérivée, classe fille, classe enfant ou sous-classe) des caractéristiques – attributs et méthodes – de la déclaration d'une autre classe (appelée classe de base, classe mère, classe parent ou super-classe)*

Lorsqu'une classe fille hérite d'une classe mère, elle peut alors utiliser les caractéristiques de sa classe mère, mais **conserve la possibilité de posséder ses propres attributs et méthodes** (en complément)

→ *les membres de la classe mère doivent posséder des niveaux de visibilité compatibles (logiquement "Protected")*

Polymorphisme : capacité de l'objet à posséder plusieurs formes

(dérive directement du principe d'héritage)

Un objet hérite des attributs et méthodes de ses ancêtres, mais garde toujours la **capacité de pouvoir redéfinir une méthode afin de la réécrire (ou de la compléter)**

C'est donc la **capacité du système à choisir dynamiquement la méthode qui correspond au type réel de l'objet** en cours

→ *le comportement de l'objet devient modifiable à volonté*

Exemple : avec un objet « Véhicule » et ses descendants « Bateau », « Avion », « Voiture », possédant tous une méthode « Avancer », le système appellera la fonction « Avancer » spécifique suivant que le véhicule est un « Bateau », un « Avion » ou bien une « Voiture »

Classes abstraites :

Une classe abstraite est une **classe dont l'implémentation n'est pas complète et qui n'est pas instanciable** (en objet)

Elle sert de **base à d'autres classes dérivées (héritées)**

Le mécanisme des classes abstraites permet de définir des comportements (méthodes) dont **l'implémentation (le code dans les méthodes) se fait dans les classes filles**

Ainsi, on a l'assurance que les classes filles respecteront le contrat défini par la classe mère abstraite

→ *ce contrat est une interface de programmation*

Interfaces :

Une interface est donc une **classe abstraite qui ne contient que des méthodes abstraites et n'implémente aucun attribut** (hormis des constantes), c'est-à-dire des **méthodes sans implémentation** (sans code)

Elle sert de **formalisme pour les classes qui implémentent cette dernière** (par héritage)

Les méthodes abstraites de l'interface ne sont accessibles qu'à partir des classes qui implémentent l'interface et ces classes doivent les redéfinir en les implémentant

→ *l'objectif est de fournir une porte d'accès à une fonctionnalité en cachant les détails de sa mise en œuvre (propre à chaque classe dérivée)*

II. Introduction à la POO

II. Introduction à la POO

1) Historique et comparaison
2) Impossibilités et enjeux : passer du procédural à l'objet
3) Syntaxe rapide généraliste de plusieurs langages accueillant l'objet

1) Historique et comparaison

Le langage **Simula 67**, en implantant les Record Class de Hoare, pose les constructions qui seront celles des langages orientés objets à classes : classe, polymorphisme, héritage,...

Mais c'est réellement avec **Smalltalk 71** (puis Smalltalk 80), inspiré en grande partie par Simula 67 et Lisp, que les principes de la programmation par objets, résultat des travaux d'Alan Kay, sont véhiculés : objet, encapsulation, messages, typage et polymorphisme

A partir des **années 1980**, commence l'effervescence des langages à objets : **C++ (1983), Objective-C (1984), Eiffel (1986), Common Lisp Object System (1988),...**

Les **années 1990** voient l'âge d'or de l'extension de la programmation par objets dans les différents secteurs du développement logiciel

2) Impossibilités et enjeux : passer du procédural à l'objet

Programmation procédurale : → *linéaire, difficilement interactive*

On s'intéresse à écrire les **étapes séquentielles nécessaires** pour résoudre un problème

Met l'accent sur les **étapes pour réaliser une tâche**

Le programme est la **liste des tâches et des opérations à exécuter**

Convient mieux aux **applications ne nécessitant pas ou peu d'interaction avec les utilisateurs**

Principaux problèmes :

- **Difficulté de réutilisation** du code
- Critères de **qualité facilement violés** : modularité, lisibilité,...
- Danger du **"code spaghetti"**
- **Difficulté de la maintenance** des grandes applications

Programmation orientée objets : → *non linéaire, interactive*

On s'intéresse à **modéliser le problème par un ensemble d'objets**

Met l'accent sur les **objets requis pour résoudre un problème**

Le programme est l'**ensemble des objets et des interactions entre ces objets**

En programmation orientée objets, on cherche à **identifier les objets impliqués et leurs responsabilités respectives** → *UML*

3) Syntaxe rapide généraliste de plusieurs langages accueillant l'objet

Exemple de classe en C++ :

```
class Point
{
private:
  int x;
  int y;

public:
  Point(int x, int y) : x(x), y(y) {}

  int getX() const { return x; }
  int getY() const { return y; }

  bool isOrigin() const { return x == 0 && y == 0; }
  Point translate(const Point& point) const {
    return Point(x + point.getX(), y + point.getY());
  }
};
```

Exemple de classe en Java :

```
public class Point {
  private final int x;
  private final int y;

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

  public int getX() { return x; }
  public int getY() { return y; }

  public boolean isOrigin() { return (x == 0) && (y == 0); }

  public Point translate(Point point) {
    return new Point(x + point.x, y + point.y);
  }
}
```

Exemple de classe en PHP :

```
class Point {
  private $x;
  private $y;

  public function __construct($x, $y) {
    $this->x = (int)$x;
    $this->y = (int)$y;
  }

  public function getX() { return $this->x; }
  public function getY() { return $this->y; }

  public function isOrigin() { return ($this->x == 0) && ($this->y == 0); }

  public function translate(Point $point) {
    return new Point($this->x + $point->x, $this->y + $point->y);
  }
}
```

Exemple de classe en Python :

```
class Point(object):
    def __init__(self, x, y):
        super(Point, self).__init__()
        self._x = x
        self._y = y

    @property
    def x(self):
        return self._x
    @property
    def y(self):
        return self._y

    def is_origin(self):
        return (self._x == 0) and (self._y == 0)

    def translate(self, point):
        return Point(self._x + point.x, self._y + point.y)
```

III. Utiliser l'objet

III. Utiliser l'objet

1) Les instances des objets
2) Staticité et dynamicité : correspondance avec la vie réelle

1) Les instances des objets

Rappel :

Une classe permet de décrire une catégorie d'objets

L'ensemble des objets de cette catégorie sont les instances de la classe

Techniquement :

En général, **l'instanciation d'une classe passe par l'utilisation du mot réservé « new »**. On peut ensuite utiliser l'objet instancié et invoquer les méthodes de la classe

Exemple en Java :

```
public class Point {
   //Variable de classe (ne dépend d'aucun objet)
   public static double dimension = 2;

   //Variables d'instance (attributs d'un objet)
   private double x;
   private double y;

   //Constructeur par défaut
   public Point() {
      this(0,0);
   }

   //Constructeur avec arguments
   public Point(double x , double y) {
      this.x = x;
      this.y = y;
   }
```

→ *chaînage des constructeurs : le constructeur par défaut fait appel au constructeur avec arguments par « this(0,0) » avec les arguments 0 et 0*

```
  //Accesseurs de l'attribut x
  public double getX() {
    return this.x;
  }
  public void setX(double x) {
    this.x = x;
  }
  //Accesseurs de l'attribut y
  public double getY() {
    return this.y;
  }
  public void setY(double y) {
    this.y = y;
  }
  //Méthodes d'instance (d'un objet)
  public void symetrieSelonX() {
    this.y = -this.y;
  }
  public void symetrieSelonY() {
    this.x = -this.x;
  }
  //Méthode de classe (ne dépend d'aucun objet)
  public static double quelleDimension() {
    return dimension;
  }
}
```

```
public class ExemplePoint {
  public static void main(String args[]) {
    //On crée un Point
    Point monPoint = new Point(4,6);

    //On appelle la méthode symetrieSelonX() sur l'instance monPoint de Point
    monPoint.symetrieSelonX();
    //Les accesseurs sont appelés comme des méthodes d'instance normales
    System.out.println("L'abscisse de monPoint est : " + monPoint.getX());
    System.out.println("L'ordonnée de monPoint est : " + monPoint.getY());
    //Appel de la méthode de classe quelleDimension()
    System.out.println("La dimension est : " + Point.quelleDimension());
  }
}
```

Constructeur par défaut : Point monPoint = new Point();

→ *monPoint a pour coordonnées (0,0)*

La création d'une instance se fait de cette manière, Point est devenu un nouveau type de donnée, on a créé un objet

2) Staticité et dynamicité : correspondance avec la vie réelle

Les propriétés statiques se rapportent à la classe et sont communes à tous les objets de la classe :

- Constantes
- Attributs statiques (variables de classe)
- Méthodes statiques (méthodes de classe dont les constructeurs)

Les propriétés dynamiques se rapportent à une instance particulière de la classe (objet) :

- Variables d'instance (attributs d'un objet)
- Méthodes d'instance (opérations d'un objet)

Un attribut peut être statique (précédé de « static ») :

- Il n'existe qu'un exemplaire de l'attribut

- L'initialisation a lieu à la première référence à la classe

- Il est accessible directement à l'aide du nom de la classe

Un attribut peut être dynamique (par défaut) :

- Chaque instance a son propre exemplaire

- Il existe des valeurs par défaut mais on préférera l'initialiser lors de la déclaration ou avec les constructeurs

Une méthode peut être statique (précédée de « static » ou constructeur) :

- Elle est invocable à l'aide du nom de la classe :

« nomClasse.nomMethode(arguments) »

- Elle ne peut accéder aux variables non statiques

Une méthode peut être dynamique (par défaut) :

- Elle est invocable depuis un objet d'une autre classe à l'aide du nom d'une référence : « objetReveceur.nomMethode(arguments) »

- Elle est invocable depuis un objet de la classe courante par « nomMethode(arguments) »

- Dans le corps, on pourra désigner l'objet receveur par « this »

- Elle peut accéder / modifier les attributs

IV. Eléments principaux d'architecture

IV. Eléments principaux d'architecture

1) Les classes, attributs et méthodes : éléments fondamentaux
2) Visibilité : comment, pourquoi
3) Héritage et réutilisation du code
4) Interfaces et abstraction : préparation raisonnée d'une architecture

1) Les classes, attributs et méthodes : éléments fondamentaux

```
public class Point {
   //Variable de classe (ne dépend d'aucun objet)
   public static double dimension = 2;

   //Variables d'instance (attributs d'un objet)
   private double x;
   private double y;

   //Constructeur par défaut
   public Point() {
      this(0,0);
   }

   //Constructeur avec arguments
   public Point(double x , double y) {
      this.x = x;
      this.y = y;
   }
```

```
  //Accesseurs de l'attribut x
  public double getX() {
    return this.x;
  }
  public void setX(double x) {
    this.x = x;
  }
  //Accesseurs de l'attribut y
  public double getY() {
    return this.y;
  }
  public void setY(double y) {
    this.y = y;
  }
  //Méthodes d'instance (d'un objet)
  public void symetrieSelonX() {
    this.y = -this.y;
  }
  public void symetrieSelonY() {
    this.x = -this.x;
  }
  //Méthode de classe (ne dépend d'aucun objet)
  public static double quelleDimension() {
    return dimension;
  }
}
```

2) Visibilité : comment, pourquoi

Les mots-clés concernant l'accessibilité sont les suivants (en Java) :

- Aucun mot-clé : accessible par les classes du même package

- Public : accessible par toutes les classes

- Protected : accessible par toutes les classes héritées et les classes du même package, inaccessible par les autres

- Private : inaccessible par toute autre classe

→ *ces termes sont utilisables sur les variables et les méthodes*

Rappel : le terme « static » permet aussi de spécifier des variables de classe ou des méthodes de classe

3) Héritage et réutilisation du code

Exemple en Java :

```
public class Vehicule {
   protected String marque, modele, couleur;

   public Vehicule(String marq, String mod, String coul) {
      this.marque = marq;
      this.modele = mod;
      this.couleur = coul;
   }

   public void Affiche() {
      System.out.println("Marque : " + this.marque);
      System.out.println("Modèle : " + this.modele);
      System.out.println("Couleur : " + this.couleur);
   }
}
```

```
public class Moto extends Vehicule {
   public Moto(String marq, String mod, String coul) {
      super(marq, mod, coul);
   }
}
public class Camion extends Vehicule {
   public Camion(String marq, String mod, String coul) {
      super(marq, mod, coul);
   }
}
public class Auto extends Vehicule {
   public Auto(String marq, String mod, String coul) {
      super(marq, mod, coul);
   }
}
public class Test {
   public static void main(String[] args) {
      Vehicule v1 = new Vehicule("Renault", "Twingo", "bleue");
      Auto v2 = new Auto("Peugeot", "504", "blanche");
      Moto v3 = new Moto("Honda", "CB500", "blanche");
      Camion v4 = new Camion("Berliet", "Camion Benne", "gris");
      v1.Affiche();
      v2.Affiche();
      v3.Affiche();
      v4.Affiche();
   }
}
```

4) Interfaces et abstraction : préparation raisonnée d'une architecture

Interfaces (en Java) :

Une interface est un **type comme une classe mais abstrait, qui ne peut donc pas être instancié** (par « new » et constructeurs)

Décrit un **ensemble de signatures de méthodes, sans implémentation**, qui doivent être implémentées dans toutes les classes qui l'implémentent

→ *son concept réside dans le regroupement de plusieurs classes implémentant un ensemble commun de méthodes (sous un même type)*

- Elle contient des **signatures de méthodes, mais pas de variables**

- Elle **peut hériter d'une autre interface** (par « extends »)

- **Une classe** (abstraite ou non) **peut implémenter plusieurs interfaces** (séparées par des virgules, après le mot-clé « implements »)

Exemple : → *interface « Forme » décrivant les méthodes qui doivent être implémentées par les classes « Rectangle » et « Cercle »*

```
public interface Forme {
   public int surface();
   public void affiche();
}

public class Rectangle implements Forme {
   ...
}
public class Cercle implements Forme {
   ...
}
```

(si une classe implémente une interface sans implémentation de toutes ses méthodes, une erreur de compilation se produit, sauf si la classe est une classe abstraite)

Classes abstraites (en Java) :

Le concept de classe abstraite **se situe entre celui de classe et celui d'interface**

C'est une **classe qu'on ne peut pas directement instancier car certaines de ses méthodes ne sont pas implémentées**

Elle peut donc **contenir des variables, des méthodes implémentées et des signatures de méthodes à implémenter**

Elle peut implémenter – partiellement ou totalement – des interfaces et hériter d'une classe ou d'une autre classe abstraite

→ *le mot-clé « abstract » est utilisé devant le mot-clé « class » pour déclarer une classe abstraite, ainsi que pour la déclaration des signatures des méthodes à implémenter*

Exemple : → *une interface ne peut contenir de variables, il faut transformer « Forme » en classe abstraite*

```
public abstract class Forme {
   protected int origine_x;
   protected int origine_y;
   public Forme() {
      this.origine_x = 0;
      this.origine_y = 0;
   }
   public int getOrigineX() {
      return this.origine_x;
   }
   public int getOrigineY() {
      return this.origine_y;
   }
   public void setOrigineX(int x) {
      this.origine_x = x;
   }
   public void setOrigineY(int y) {
      this.origine_y = y;
   }
   public abstract int surface();
   public abstract void affiche();
}
```

→ il faut également rétablir l'héritage des classes « Rectangle » et « Cercle » vers « Forme »

```
public class Rectangle extends Forme {
   ...
}

public class Cercle extends Forme {
   ...
}
```

(lorsqu'une classe hérite d'une classe abstraite, elle doit soit implémenter les méthodes abstraites de sa super-classe en les dotant d'un corps, soit être elle-même abstraite si au moins une des méthodes abstraites de sa super-classe reste abstraite)