# TP - Héritage & Polymorphisme
# Gestion de courbes
*version 3.0*

## Table des matières



**Avertissement :**
— Ce document est en version 3.0. Merci de signaler toutes erreurs rencontrées.

# 1 Objectifs

— Approfondir les notions de classes et d'objets
— Comprendre le concept d'héritage et de polymorphisme
— Savoir implémenter et utiliser une hiérarchie de classes, des constructeurs et des méthodes polymorphes

Ce TP sert à illustrer le concept d'héritage. L'intérêt de ce concept et sa mise en œuvre seront illustrés dans le cadre de ce TP. L'application concernera la gestion d'une collection de courbes qui sera ensuite exploitée dans les prochains TP dédiés à la réalisation d'une interface graphique.

# 2 Cahier des charges

Ce TP a pour but de permettre de :
— calculer la longueur et l'aire de différents types de courbes
— manipuler différents types de courbes : cercles, polygones simples (c'est à dire sans intersections d'arêtes) et polygones réguliers
— tirer profit de la programmation orientée objet pour rendre le code plus efficace et plus extensible (par exemple pour facilement rajouter d'autres types de courbes sans toucher le reste du code)

# 3 Mise en place

1. Créez un dossier `TP_heritage` qui sera votre dossier de travail [1].
2. Écrivez les classes `Cercle` et `Polygone` [2] et leur constructeur respectif. Un cercle est défini par son centre et son rayon. Un polygone est défini par un tableau de points dont chaque élément décrit un de ses sommets. Pour le centre du cercle et les sommets du polygone, vous avez à votre disposition la classe `APoint`. Téléchargez le fichier `APoint.java` dans votre dossier de travail et compilez le [3].
3. Écrivez les méthodes `longueur()` dans chacune de ces classes qui calcule la longueur (ou périmètre dans le cas des courbes fermées) de la courbe. On rappelle que la constante $\pi$ est définie dans `java` par `Math.PI`.
4. Complétez la classe `GestionCourbe` [4] qui contient la méthode `main` permettant de tester ces deux classes. Cette méthode doit créer des cercles et des polygones et afficher dans le terminal la longueur de chaque courbe (figure 1). Pour cela, on pourra ajouter la méthode `toString()` dans les classes `Cercle` et `Polygone` (vous pouvez vous inspirer de la classe `APoint`).

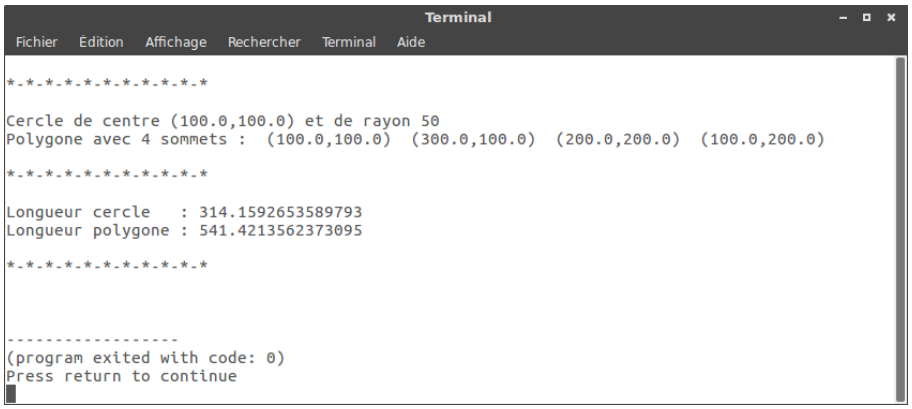


FIGURE 1 – Exemple d'affichage suite à l'exécution de la classe `GestionCourbe`

1. Pour faire ce TP en ligne sous Caseine, cliquez sur ce **lien**.
2. Sous Caseine, il faut créer un nouveau fichier, pour cela cliquez sur le symbole `+` et sur l'icône `nouveau`.
3. Sous Caseine, il n'y a rien à faire, la classe `APoint` fait déjà partie du TP.
4. Sous Caseine, le squelette de cette classe est déjà fournie, il suffit de le mettre à jour.

# 4 Concept d'héritage - Gestion de différents types de courbes

Pour l'instant, vos classes peuvent être représentées par les diagrammes UML ci dessous (figure 2).

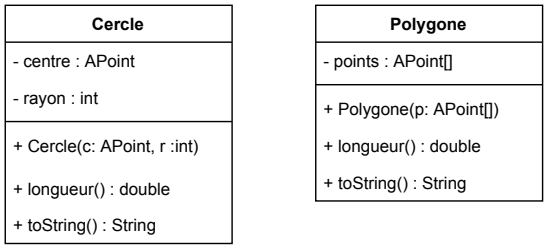


FIGURE 2 – Diagrammes UML des classes `Cercle` et `Polygone`

## 4.1 Mise à jour du diagramme UML

On souhaite maintenant pouvoir manipuler un tableau contenant à la fois des cercles et des polygones.

1. Quelle hiérarchie de classes permet de rendre cela possible ?
2. Quelle(s) méthode(s) doit(doivent) être déclarée(s) en tant qu'abstraite(s) ?
3. Proposer un diagramme UML correspondant à cette hiérarchie.

## 4.2 Mise en œuvre de l'héritage

Une fois le diagramme UML validé, votre programme doit pouvoir maintenant gérer plusieurs courbes. Pour cela nous allons utiliser un tableau pour stocker les courbes.

Voici le code de la classe `GestionCourbe` que vous allez maintenant utiliser :

```
public class GestionCourbe{

   public static void main(String[] args){

      // Création d'un tableau de 5 courbes
      Courbe[] tabCourbe = new Courbe[5];

      // création de 3 cercles et 2 polygones
      // ... code à compléter ...

      // Remplissage du tableau de courbes
      tabCourbe[0]=c1;
      tabCourbe[1]=c2;
      tabCourbe[2]=c3;
      tabCourbe[3]=poly1;
      tabCourbe[4]=poly2;

      // Affichage des données des courbes
      for (int i =0;i<tabCourbe.length;i++){
         System.out.println(tabCourbe[i]);
      }

   }
}
```

1. Modifiez votre code pour prendre en compte cette hiérarchie de classes.
2. En vous inspirant du code ci dessus, créez 3 cercles et 2 polygones de votre choix et stockez les dans le tableau de courbes.

3. Affichez dans le terminal les caractéristiques et les longueurs de tous les éléments de ce tableau.
4. Calculez et affichez la moyenne des longueurs des courbes.

### 4.3 Compléments d'algorithmie et d'héritage

Votre programme permet de gérer différents types de courbes. Nous souhaitons le compléter en calculant maintenant l'aire et la position du barycentre de chaque élément du tableau de courbes.

1. En vous inspirant de la méthode `longueur()`, ajoutez une méthode `aire()` qui renvoie l'aire de la courbe concernée. Pour calculer l'aire $A$ d'un polygone simple (c'est à dire sans intersections d'arêtes), on pourra utiliser la formule suivante :

$$A = \frac{1}{2}\sum_{i=0}^{n}(x_i y_{i+1} - x_{i+1}y_i)$$

*Attention, ce calcul nécessite de définir les points en respectant le sens trigonométrique.*

2. De la même manière, ajoutez maintenant la méthode `barycentre()` qui renvoie le barycentre de la courbe concernée. Pour calculer les coordonnées $[x_G, y_G]$ du barycentre $G$ d'un polygone simple, on pourra utiliser les formules suivantes :

$$x_G = \frac{1}{6A}\sum_{i=0}^{n}(x_i + x_{i+1})(x_i y_{i+1} - x_{i+1}y_i)$$

$$y_G = \frac{1}{6A}\sum_{i=0}^{n}(y_i + y_{i+1})(x_i y_{i+1} - x_{i+1}y_i)$$

où $A$ est l'aire du polygone.

3. Mettez à jour la classe `GestionCourbe` de manière à afficher dans le terminal, en plus des informations précédentes, l'aire de chaque courbe ainsi que la moyenne des aires de toutes les courbes.

### 4.4 Bonus

1. Calculer et afficher la distance (*i.e.* la distance euclidienne entre les barycentres des courbes concernées) entre deux courbes consécutives.
2. Calculer et afficher la distance entre chaque courbe.

## 5 Ajoutons de la couleur

Pour égayer toutes ces courbes, on offre à l'utilisateur la possibilité d'affecter une couleur à chaque courbe lors de leur construction. En `java`, il existe différentes manières de gérer la couleur. La plus simple est d'utiliser la classe `Color` qui est fourni par défaut avec `java` (lien vers la javadoc). Pour cela, il suffit de rajouter dans l'entête (c'est à dire avant `public class maClasse`) de vos classes concernées l'instruction suivante :

```
import java.awt.Color;
```

Les couleurs prédéfinies dans la classe `Color` sont : `black`, `blue`, `cyan`, `darkGray`, `gray`, `green`, `lightGray`, `magenta`, `orange`, `pink`, `red`, `white` et `yellow`. Le code RGB (proportions de rouge, vert et bleu données entre 0 et 255) de ces couleurs est donné dans le tableau ci dessous :

| Couleur | Rouge | Vert | Bleu |
|---|---|---|---|
| `black` | 0 | 0 | 0 |
| `blue` | 0 | 0 | 255 |
| `cyan` | 0 | 255 | 255 |
| `darkGray` | 64 | 64 | 64 |
| `gray` | 128 | 128 | 128 |
| `green` | 0 | 255 | 0 |
| `lightGray` | 192 | 192 | 192 |

| Couleur | Rouge | Vert | Bleu |
|---|---|---|---|
| `magenta` | 255 | 0 | 255 |
| `orange` | 255 | 200 | 0 |
| `pink` | 255 | 175 | 175 |
| `red` | 255 | 0 | 0 |
| `white` | 255 | 255 | 255 |
| `yellow` | 255 | 255 | 0 |

Par exemple, l'instruction suivante :

```
Color maCouleur = Color.black;
```

permet de déclarer la variable `maCouleur` comme un type `Color` et de lui attribuer la couleur noire.

Tout en tirant profit de la Programmation Orientée Objet et en particulier de l'héritage, les classes vont être modifiées pour prendre en compte le fait qu'une couleur peut être associée à une courbe.

1. Lors de la création d'une courbe, il faut avoir la possibilité de définir la couleur de la courbe. Si aucune couleur n'est donnée en paramètre, nous souhaitons que la couleur noire soit associée par défaut à une courbe. Quelles classes faut il modifier ? Faut il créer de nouveaux attributs ? Mettez à jour votre diagramme UML suite à ces modifications.
2. Modifiez la classe principale de manière à ne pas affecter de couleur aux deux premières courbes (elles seront donc noires) et 3 couleurs différentes de votre choix aux 3 suivantes.
3. On souhaiterait également faire apparaître dans le message affiché dans le terminal la couleur des courbes. Quelle(s) méthode(s) `toString()` faut il modifier ? Faites les modifications en conséquence.

*Remarque.* Pour la dernière question, il faut au maximum tirer profit de la POO et donc factoriser votre code en mettant dans la méthode de l'ancêtre toutes les informations communes aux différentes courbes et utiliser `super.toString()`.

*Remarque.* En `java`, lors de l'appel à la méthode `toString()` de la classe `Color`, le code RGB de la couleur concernée est affiché c'est à dire la proportion de rouge, de vert et de bleu (entre 0 et 255). On se contentera de ce type d'affichage.

# 6 Ajoutons un nouveau descendant : le polygone régulier

Dans cette partie, nous allons gérer les polygones réguliers qui sont un cas particulier des polygones. On ne considérera ici que les polygones réguliers convexes. Pour rappel, un polygone régulier est un polygone dont toutes les arêtes ont la même longueur. La figure 3 présente les principales caractéristiques d'un polygone régulier.

Pour un polygone régulier, les coordonnées de chaque sommet du polygone peuvent être calculées ainsi :

$$\begin{cases} x_i &= r\cos(i\alpha+\beta)+x_C \\ y_i &= r\sin(i\alpha+\beta)+y_C \end{cases} \quad \text{avec } i \in [0,n]$$

*Remarque.* Les angles $\alpha$ et $\beta$ sont en radians.

1. Écrivez la classe `PolygoneRegulier` qui est une classe fille de la classe `Polygone`. **Aucun attribut supplémentaire ne doit être ajouté à cette classe**. Le constructeur de cette nouvelle classe aura comme signature :
   ```
   public PolygoneRegulier(APoint centre, APoint unSommet, int nCotes)
   ```

   *Remarque.* Pour calculer l'angle $\beta$, on fera appel à la méthode `Math.atan2(double y, double x)` qui donne un résultat dans l'intervalle $[-\pi,\pi]$ (lien vers la javadoc).

   *Remarque.* Attention chaque classe `java` a implicitement un constructeur sans paramètres d'entrée et sans instructions **sauf** si vous définissez votre propre constructeur. Automatiquement `java` ajoute à tous vos constructeurs l'instruction `super()` sur la première ligne (qui donc fait appel au constructeur sans paramètres de la classe mère) **sauf** si vous avez une instruction qui fait appel à un constructeur spécifique de la classe mère. Ainsi, si vous devez appeler le constructeur d'une classe mère, cet appel doit impérativement être la première instruction du constructeur de votre classe. Dans certains cas, il peut être donc être nécessaire de créer un constructeur sans paramètres d'entrée (et éventuellement sans instructions) dans la classe mère car `java` ne l'ajoute pas automatiquement dès qu'un autre constructeur est défini. Ce constructeur peut être nécessaire pour utiliser le constructeur d'un descendant de cette classe.

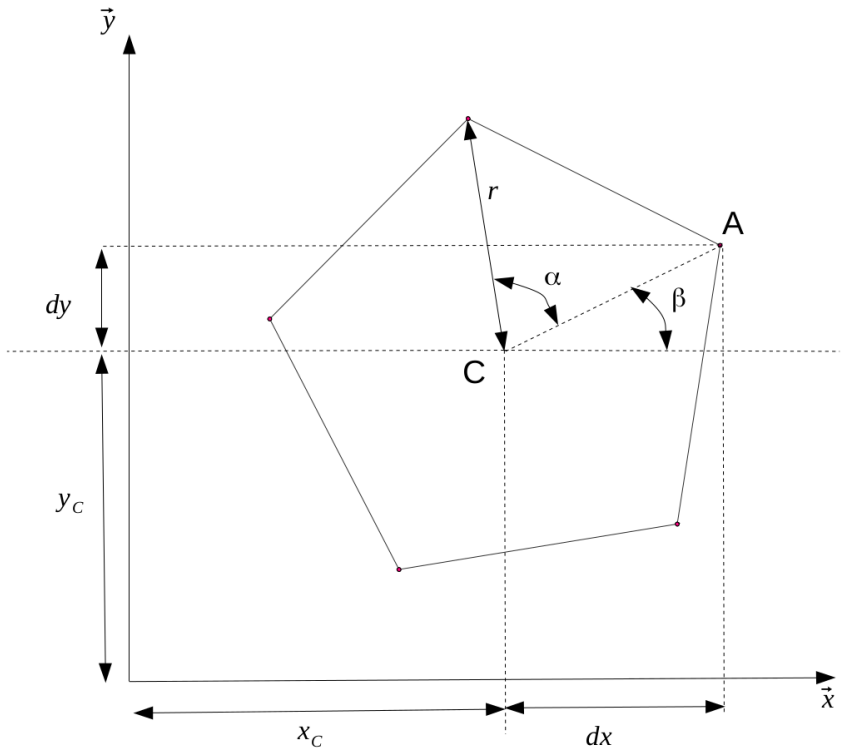


FIGURE 3 – Principales caractéristiques d'un polygone régulier

2. Dans cette classe, est il nécessaire d'implémenter les méthodes `longueur()` et `aire()` ?
3. Dans le but de minimiser le temps de calcul, il est possible de redéfinir les méthodes `longueur()` et `aire()`. Écrivez ces méthodes sachant que l'aire $A$ d'un polygone régulier de $n$ côtés de longueur $a$ se calcule ainsi :

$$A = \frac{na^2}{4\tan\left(\frac{\pi}{n}\right)}$$

# 7 Bonus - Notion d'interface

Votre programme évolue avec l'introduction d'un nouveau type de courbe : les lignes polygonales. Une ligne polygonale est décrite par un tableau de points consécutifs qui décrivent une courbe ouverte (figure 4).

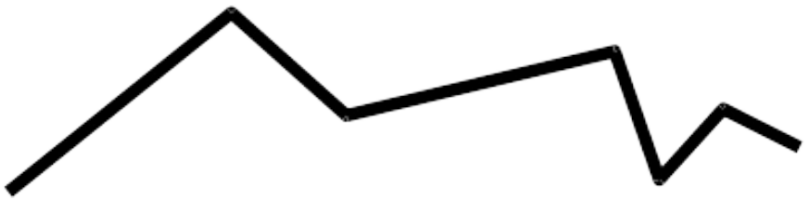

FIGURE 4 – Exemple d'une ligne polygonale décrivant une courbe ouverte

Comme pour les classes `Cercle` et `Polygone`, il est possible de calculer la longueur et le barycentre d'une ligne polygonale. Une ligne polygonale peut donc hérité directement de la classe `Courbe`.

Par contre, contrairement aux classes `Cercle` et `Polygone`, il est impossible de calculer une aire pour une courbe ouverte comme la ligne polygonale.

Proposez un nouveau diagramme UML prenant en compte ce nouveau descendant avec ses particularités.