# 1 Classe Point

## 1.1 Point en coordonnées cartésiennes

On souhaite réaliser un certain nombre de manipulations élémentaires sur les points du plan 2D. Un point sera représenté par ses coordonnées cartésiennes, abscisse `x` et ordonnée `y`.

Vous pourrez utiliser les méthodes de la classe `Math`, dont la documentation est disponible à l'adresse :
`https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/Math.html`
En voici un extrait utile :

— `Math.pow(a,b)` pour calculer $a^b$,
— `Math.sqrt(x)` pour calculer $\sqrt{x}$,
— `Math.atan2(double y, double x)` pour calculer l'angle de la représentation polaire du point $(x, y)$.
— `Math.cos(theta)` et `Math.sin(theta)` pour calculer le cosinus et le sinus d'un angle.

**Tâche 1 :** Définir une classe `CartesianPoint` avec deux attributs pour les coordonnées cartésiennes et un constructeur adapté.

**Tâche 2 :** Écrire une méthode `CartesianPoint translate(CartesianPoint q)` qui crée un nouveau point obtenu par la translation par le vecteur $\overrightarrow{Oq}$. Ainsi `p.translate(q)` sera le point de coordonnées $(0, 3)$, `p` et `q` ne doivent pas être modifiés.

**Tâche 3 :** Écrire une méthode Écrire une méthode `public String toString()` dans `CartesianPoint`, qui retourne une chaîne de caractères représentant le point. Par exemple `p.toString()` devra retourner la chaîne de caractères `"(x = 2, y = -1)"`. Attention, cette méthode n'est pas comme `display()`, rien ne doit s'afficher en console !

**Tâche 4 :** Dans la méthode `main` de la classe `Main`, écrire les instructions pour créer les points `p`, `q` et le résultat de la translation de `p` par `q`. Puis ajouter des instructions pour afficher ces trois points. Pour cela, vous pouvez profiter du fait que Java utilise la méthode `toString` pour convertir un objet en chaîne de caractères automatiquement, vous pouvez donc écrire :

```
System.out.println("p = " + p);
```

Vérifier que la méthode `translate` est correcte.

## 1.2 Points en coordonnées polaires

On voudrait maintenant représenter un point à l'aide de ses coordonnées polaires $(r, \theta)$. On rappelle la relation entre les coordonnées cartésiennes $(x, y)$ et polaires $(r, \theta)$ :

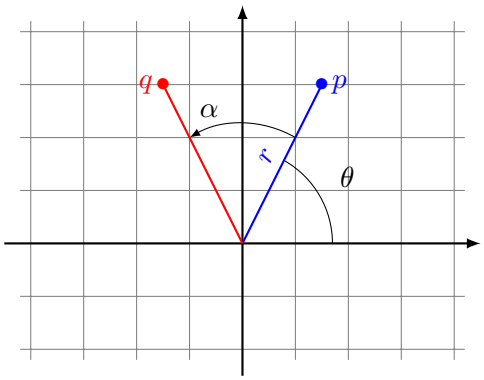
$$x = r\cos\theta \text{ et } y = r\sin\theta$$

$$r = \sqrt{x^2 + y^2} \text{ et } \theta = \arctan\frac{y}{x}$$

Tous les angles considérés seront exprimés en radians.

**Tâche 5 :** Écrire une classe `PolarPoint` qui implémente la représentation polaire d'un point du plan.

**Tâche 6 :** Dans la classe `PolarPoint`, écrire une méthode `PolarPoint rotate(double alpha)` qui permet de calculer l'image du point par une rotation autour de l'origine, d'angle $\alpha$.




**Tâche 7 :** Ajouter une méthode `public String toString()` à `PolarCoordinate`. L'appel `s.toString()` devra retourner `"(r = 2, theta = 0.7853981633974483)"`.

**Tâche 8 :** Ajouter des instructions dans le `main`, pour vérifier que la méthode `rotate` fonctionne comme attendue.

**Tâche 9 :** Implémenter deux méthodes :

— `PolarPoint toPolar()` dans la classe `CartesianPoint` qui permet de passer d'un `CartesianPoint` à un `PolarPoint`

— `CartesianPoint toCartesian()` dans la classe `PolarPoint` qui permet de passer d'un `CartesianPoint` à un `PolarPoint`

**Tâche 10 :** Compléter les classes `PolarPoint` et `CartesianPoint`, de sorte que chacune implémente les deux méthodes : `translate` et `rotate`. Vous pouvez réutiliser les méthodes écrites plus tôt.

## 2 Vecteurs

On implémente maintenant une classe pour représenter les vecteurs du plan Euclidien. Ce n'est pas très différent d'un point en coordonnée cartésienne, mais nous allons y ajouter des opérations comme l'addition ou la rotation. Pour celles-ci, vous pouvez consulter la page wikipedia `https://en.wikipedia.org/wiki/Euclidean_vector` pour avoir plus de détails.

**Tâche 11 :** Créer une classe `Vector` dans un fichier `Vector.java`. Un vecteur $\vec{v}$ pouvant être représenté dans une base orthonormale $(\vec{i}, \vec{j})$ de façon unique comme $\vec{v} = x\vec{i} + y\vec{j}$ avec $x$ et $y$ réels, on utilise deux attributs `x` et `y`.

**Tâche 12 :** Ajouter un constructeur dans la classe `Vector` initialisant les deux attributs à partir de deux paramètres.

**Tâche 13 :** Ajouter une méthode `getX` et une méthode `getY` retournant chacune la valeur d'un des deux propriétés.

**Tâche 14 :** Ajouter une méthode pour l'addition de deux vecteurs. Cette méthode prend un vecteur en paramètre, et retourne un *nouveau* vecteur correspondant à la somme de `this` et du paramètre. `this` et le paramètre ne doivent pas être modifiés.

**Tâche 15 :** Ajouter une méthode pour calculer l'opposé (*opposite*) d'un vecteur $\vec{v}$ (c'est-à-dire le vecteur dont l'addition avec $\vec{v}$ donne le vecteur nul). Cette méthode n'a pas de paramètre et retourne un nouveau vecteur opposé à `this`. `this` ne doit pas être modifié.

**Tâche 16 :** Ajouter une méthode pour la soustraction d'un vecteur à un autre. Procéder comme pour l'addition.

**Tâche 17 :** Ajouter une méthode pour la multiplication par un scalaire. C'est l'opération permettant d'obtenir un vecteur de même direction, en multipliant sa longueur par un certain ratio (par exemple pour doubler ou tripler sa longueur). Ce ratio est un double pris en paramètre. De nouveau, on retourne un nouveau vecteur, `this` ne doit pas être modifié.

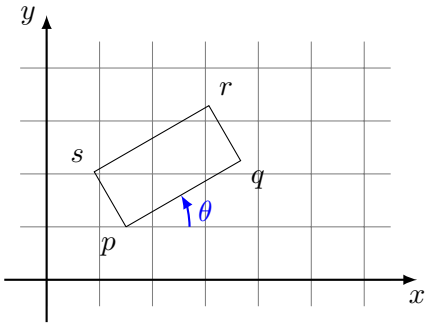
**Tâche 18 :** Ajouter une méthode pour calculer la rotation d'un vecteur par un certain angle. L'angle est en radian et est reçu en paramètre. `this` ne doit pas être modifié. On se souvient que l'image d'un vecteur de coordonnées $(x, y)$ par une rotation d'angle $\theta$ est le vecteur de coordonnées $(x\cos\theta - y\sin\theta, x\sin\theta + y\cos\theta)$.

**Tâche 19 :** Ajouter une méthode pour calculer la norme du vecteur. Vous pouvez faire usage de la méthode `Math.hypot` pour cela, consulter la documentation pour savoir comment elle fonctionne.

## 3 Classe Rectangle

Un rectangle quelconque $pqrs$ du plan Euclidien, où les points sont pris dans l'ordre trigonométrique, peut être représenté par :

— la position d'un de ses sommets $p$,
— la largeur $|pq|$,
— la hauteur $|ps|$,
— l'angle $\theta$ du côté $pq$ avec la demi-droite horizontale issue de $p$.




On utilisera la classe `Vector` pour représenter les positions des points et les vecteurs.

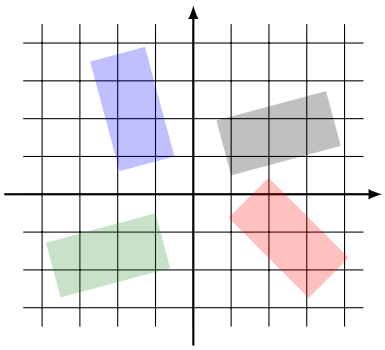
**Tâche 20 :** Créer une nouvelle classe `Rectangle` dans un nouveau fichier au nom approprié. Définir les 4 attributs nécessaires avec les types et les identifiants adéquats.

**Tâche 21 :** Ajouter un constructeur, avec 4 paramètres.

**Tâche 22 :** Ajouter quatre méthodes, une par propriété, retournant les valeurs des attributs. Les nommer avec “get” suivi du nom de l’attribut avec une majuscule.

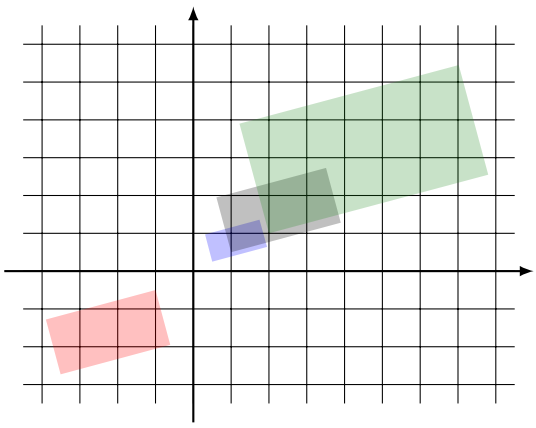
**Tâche 23 :** Ajouter une méthode de translation. Cette méthode prend un vecteur en paramètre, et retourne un nouveau rectangle, image de `this` par la translation selon le vecteur reçu. `this` ne doit pas être modifié.

**Tâche 23 :** Ajouter une méthode de rotation. Cette méthode prend un angle en radian en paramètre. Elle retourne l’image de `this` par la rotation du plan, centrée en l’origine et selon l’angle reçu en paramètre. Ce rectangle s’obtient en prenant l’image du sommet $p$ par la rotation, et en ajoutant l’angle d’inclinaison du rectangle avec l’angle de la rotation. La largeur et la hauteur sont invariantes par rotation. `this` ne doit pas être modifié.



Ici le rectangle noir subit trois rotations selon différents angles, $-\pi/3$ pour obtenir le rouge, $\pi/2$ pour obtenir le bleu, $\pi$ pour obtenir le vert.

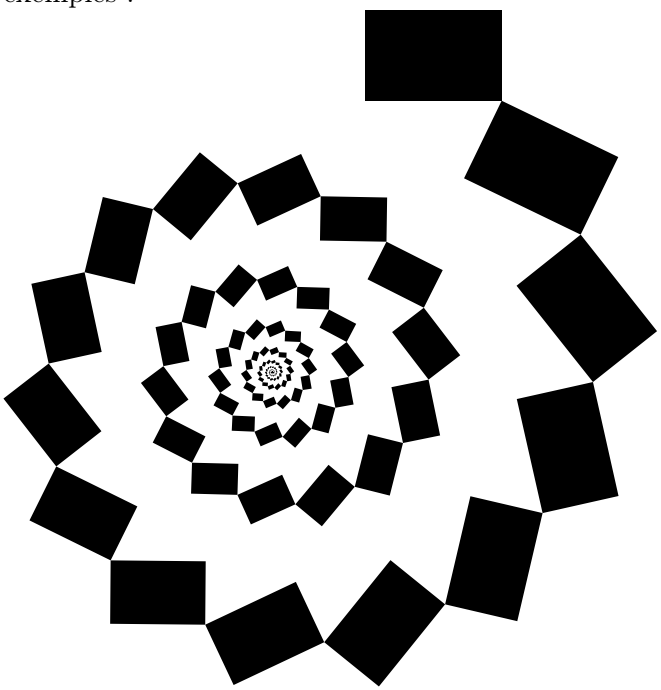
**Tâche 24 :** Ajouter une méthode de grossissement. Elle prend en paramètre un nombre décimal, et réalise une homothétie dont le centre est l’origine du repère.

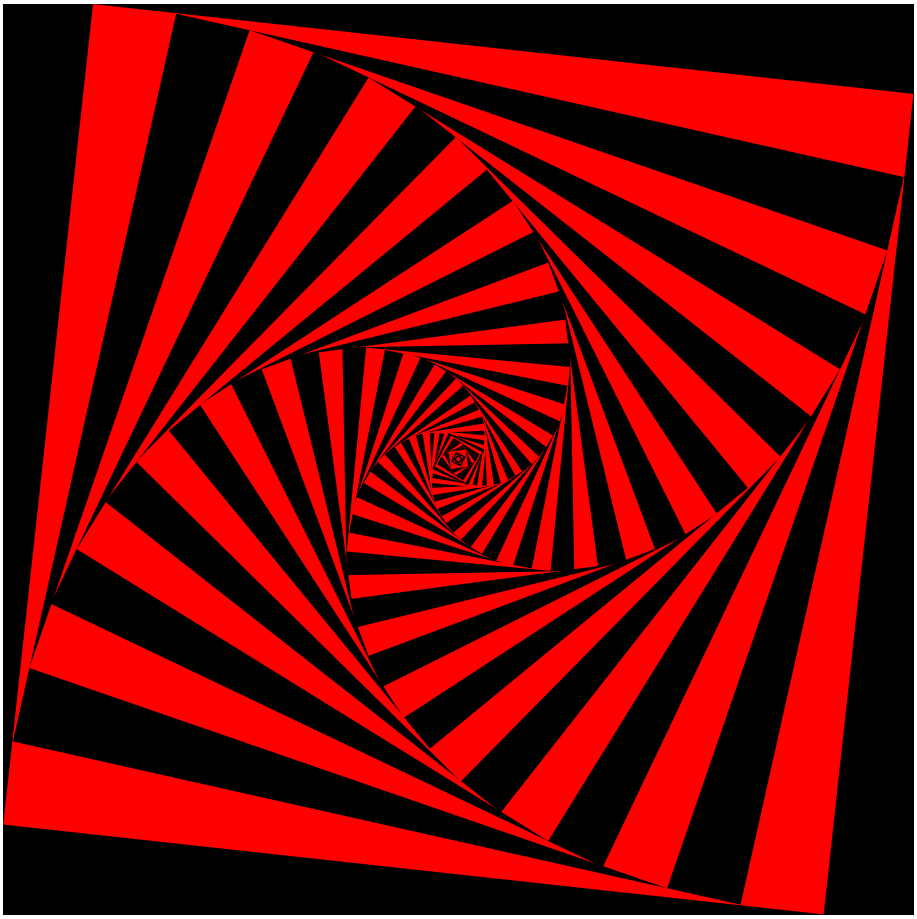


Ici, le rectangle noir subit un grossissement selon différents facteurs, -1 pour obtenir le rectangle rouge, 0.5 pour obtenir le bleu et 2 pour obtenir le vert.

## 4 Génération d'images en SVG

Grâce aux méthodes de transformations des rectangles, on peut créer des images intéressantes, en appliquant la même transformation plusieurs fois de suite sur le même rectangle, et en affichant tous les rectangles obtenus. voici deux exemples :





**Tâche 25 :** Dans la classe `Rectangle`, ajouter une méthode `toSvg()`, retournant un texte correspondant à l'encodage du rectangle au format SVG. Pour rappel, voici un exemple de rectangle au format SVG :

```
<rect x='40.0' y='30.0' width='20.0' height='20.0' transform='rotate(45.0 50.0 40.0)' />
```

La rotation est encodé avec trois paramètres : l'angle en degré, l'abscisse et l'ordonnée du centre de la rotation.

**Tâche 26 :** Créer une nouvelle classe `Main`, avec une méthode de déclaration `public static void main(String[] args)`.

**Tâche 27 :** Créer une méthode prenant un rectangle et un entier en paramètre, et ne retournant rien. Cette méthode doit afficher le rectangle au format SVG, puis effectuer une transformation sur le rectangle au format SVG, puis effectuer une transformation sur le rectangle (de votre choix), puis se rappeler elle-même avec ce nouveau rectangle. L'entier permet de contrôler le nombre de rectangle à afficher : c'est la profondeur de récursion autorisée. S'il est nul, la méthode doit terminer sans se rappeler elle-même, sinon lorsqu'elle se rappelle, elle se rappelle avec une

profondeur plus petite de 1. Ainsi le nombre de rectangles qui s'afficheront sera égal à 1 + la profondeur initiale. Pour tester si la profondeur est nulle, il faudra utiliser une structure conditionnelle de la forme `if (condition) { instructions }`.

**Tâche 28 :**Écrire les instructions de la méthode 'main, appelant votre méthode récursive.

Reportez-vous à la dernière section du TD 2, pour savoir comment générer un fichier SVG complet.

**Tâche 29 :**Essayer de reproduire l'un des dessins ci-dessus, ou une œuvre de votre composition, à partir de cette technique.