BE #2 : Modélisation de formes géométriques

Ceci est la version enseignants incluant les corrections. Ce TD opère quelques changements par rapport à l'an dernier :

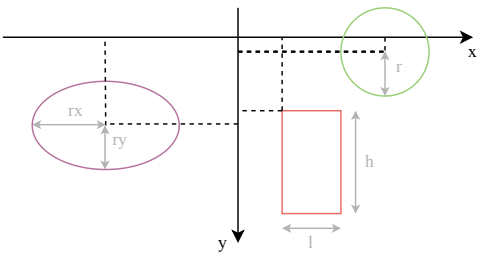
- Remplacement de **Cercle/Rectangle/Carre** par **Rectangle/Ellipse/Cercle** pour compliquer la réutilisation des solutions de l'an dernier.
- Remplacement des méthodes *perimetre* et *surface* par des méthodes un tout petit peu plus complexes (`contient_point` et `redimension_par_points`), qui pourront servir pour le BE #4.
- Modification de certaines coordonnées de double à entier pour réduire les bugs liés à l'arithmétique flottante.
- La partie dédiée à la conception de tests unitaires est réservée à une partie bonus.

Le but de ce BE est d'illustrer le concept d'héritage de la programmation objet, en concevant un module pour manipuler des formes géométriques avec Python. Vous commencerez par définir les classes et leurs attributs, puis implémenterez les méthodes, et les validerez avec des tests.

Modélisation avec UML (1h30)

Les formes géométriques sont représentées par des classes, et l'héritage sera utilisé pour factoriser les propriétés communes. Nous nous limitons à un repère à deux dimensions orthonormé, avec les axes croissant vers la droite et le bas. Les coordonnées dans ce repère sont des entiers relatifs (c'est-à-dire possiblement négatifs). Dans cet espace, nous choisissons de représenter les formes suivantes :

- Les rectangles caractérisés par leur origine (`x` , `y`) et leurs dimensions (`l` , `h`).
- Les ellipses caractérisées par leur origine (`x` , `y`) et leurs rayons aux axes (`rx` , `ry`).
- Les cercles caractérisés par leur origine (`x` , `y`) et leur rayon.



Exercice 1 - Représentez les 3 classes dans un diagramme de classes UML (*voir diagrams.net pour dessiner en ligne, avec l'onglet UML sur la gauche de l'interface*). Il est recommandé de commencer les noms des classes par une majuscule et les attributs par une minuscule. Durant tout ce BE on considèrera uniquement des attributs privés.

Rectangle
- x: int
- y: int
- l: int
- h: int

Ellipse
- x: int
- y: int
- rx: double
- ry: double

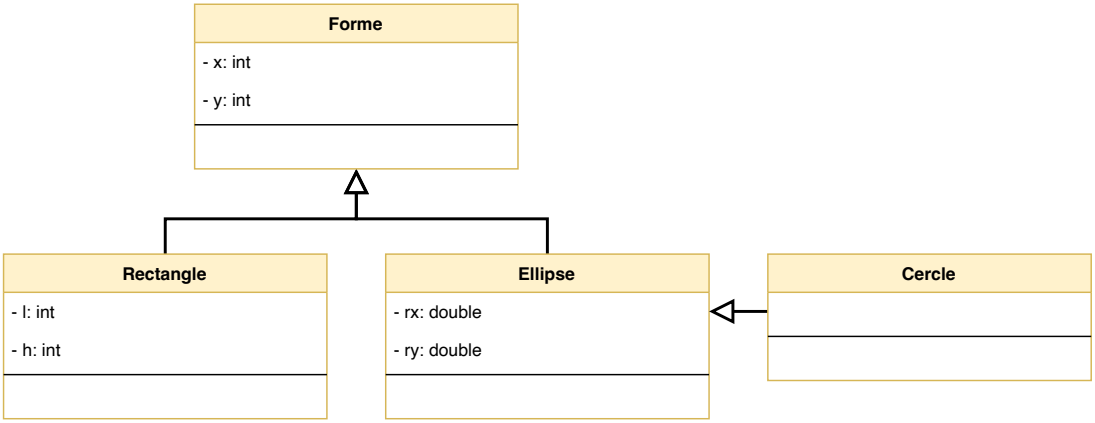
Cercle
- x: int
- y: int
- r: double

Les attributs `x` et `y` étant partagés par les trois classes et le cercle étant un cas particulier d'ellipse, on introduit l'héritage pour les regrouper. Toutes les formes géométriques hériteront d'une même classe **Forme**, et le cercle héritera de l'ellipse. L'intérêt de ces relations d'héritage est double :

- Du point de vue des développeurs du module, les méthodes dont le code est identique entre formes (ex. translation) sont fusionnées dans **Forme**, réduisant la quantité de code à produire (et donc la multiplication des erreurs possibles).
- Du point de vue des utilisateurs du module, on peut écrire du code qui manipule des rectangles et des ellipses (*p. ex.* système de collisions de formes) sans avoir à écrire du code séparément pour les rectangles et les ellipses. Cet aspect

sera illustré dans un prochain BE.

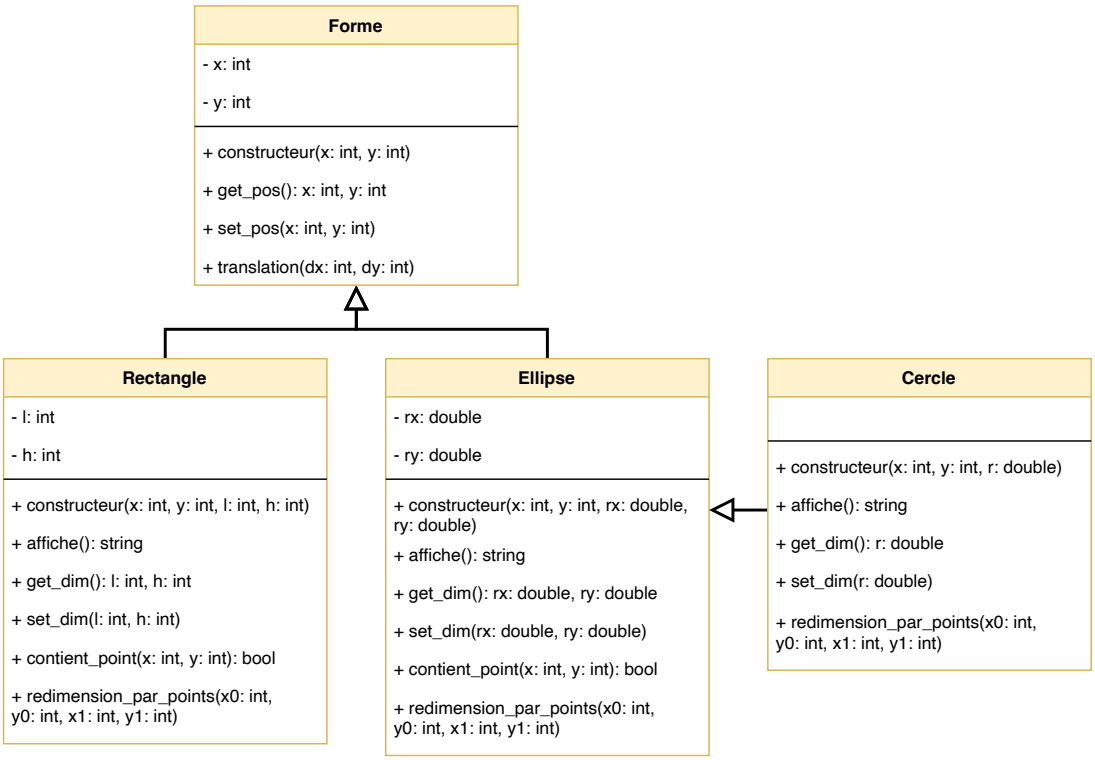
Exercice 2 - Mettez à jour le diagramme UML en incluant la classe **Forme** et les relations d'héritage. Seuls les attributs seront inclus pour le moment.



Enfin, on vous demande de supporter a minima pour chaque forme les méthodes suivantes :

- `translation(dx, dy)` , qui effectue une translation selon un vecteur donné.
- `contient_point(x, y)` , qui renvoie `True` si et seulement si le point donné est à l'intérieur de la forme ou sur sa frontière.
- `redimension_par_points(x0, y0, x1, y1)` , qui redimensionne la forme telle qu'elle soit incluse dans le rectangle de coins (`x0` , `y0`) et (`x1` , `y1`). Dans le cas du cercle, il faudra également qu'il soit le plus proche du premier coin. Cette méthode est utile par exemple dans diagrams.net pour le tracé de formes par appui-déplacement de souris.

Exercice 3 - Complétez le diagramme UML avec ces méthodes. Les constructeurs devront également être renseignés (méthode `__init__` en *Python*), ainsi que les méthodes d'accès (les fameux *getter/setter*) et d'affichage (méthode `__str__`).



Exercice 4 - Écrivez un squelette de code correspondant à votre diagramme UML, dans un fichier *formes.py*. Seuls les constructeurs devront être implémentés. À l'intérieur des autres méthodes, vous mettrez l'instruction `pass` de *Python* (qui ne fait rien mais vous rappelle que le code est inachevé).

```
class Forme:
    def __init__(self, x, y):
        self.__x = x
```

```
        self.__y = y
    def get_pos(self):
        pass
    def set_pos(self, x, y):
        pass
    def translation(self, dx, dy):
        pass

class Rectangle(Forme):
    def __init__(self, x, y, l, h):
        Forme.__init__(self, x, y)
        self.__l = l
        self.__h = h
    def __str__(self):
        pass
    def get_dim(self):
        pass
    def set_dim(self, l, h):
        pass
    def contient_point(self, x, y):
        pass
    def redimension_par_points(self, x0, y0, x1, y1):
        pass

class Ellipse(Forme):
    def __init__(self, x, y, rx, ry):
        Forme.__init__(self, x, y)
        self.__rx = rx
        self.__ry = ry
    def __str__(self):
        pass
    def get_dim(self):
        pass
    def set_dim(self, rx, ry):
        pass
    def contient_point(self, x, y):
        pass
    def redimension_par_points(self, x0, y0, x1, y1):
        pass

class Cercle(Ellipse):
    def __init__(self, x, y, r):
        Ellipse.__init__(self, x, y, r, r)
    def __str__(self):
        pass
    def get_dim(self):
        pass
    def set_dim(self, r):
        pass
    def redimension_par_points(self, x0, y0, x1, y1):
        pass
```

Implémentation des méthodes (2h30)

Créez un fichier *test_formes.py* dans le même dossier que *formes.py* et initialisé avec le code suivant :

```
from formes import *

def test_Rectangle():
    r = Rectangle(10, 20, 100, 50)
    str(r)
    assert r.contient_point(50, 50)
    assert not r.contient_point(0, 0)
    r.redimension_par_points(100, 200, 1100, 700)
    assert r.contient_point(500, 500)
    assert not r.contient_point(50, 50)

def test_Ellipse():
```

```
    e = Ellipse(60, 45, 50, 25)
    str(e)
    assert e.contient_point(50, 50)
    assert not e.contient_point(11, 21)
    e.redimension_par_points(100, 200, 1100, 700)
    assert e.contient_point(500, 500)
    assert not e.contient_point(101, 201)

def test_Cercle():
    c = Cercle(10, 20, 30)
    str(c)
    assert c.contient_point(0, 0)
    assert not c.contient_point(-19, -9)
    c.redimension_par_points(100, 200, 1100, 700)
    assert c.contient_point(500, 500)
    assert not c.contient_point(599, 500)

if __name__ == '__main__':
    test_Rectangle()
    test_Ellipse()
    test_Cercle()
```

La commande `assert` de *Python* permet de spécifier une assertion (une condition qui doit toujours être vraie) à un point du programme. Elle sert, avant un bloc de code, à en documenter les prérequis et, après un bloc de code, à en vérifier les résultats. Son échec signifie alors un bug du programme. `assert` reçoit une expression (comme ce qu'on passe à `if`), et vérifie son résultat :

- Si `True`, l'assertion est vraie donc pas de problème, `assert` ne fait rien.
- Si `False`, l'assertion est fausse donc une exception `AssertionError` est déclenchée.
- Si l'expression renvoie une autre valeur, celle-ci est convertie en booléen pour se ramener aux deux cas précédents.

La vérification de cette condition est faite une fois au moment de son exécution (l'assertion ne sera pas valide dans le reste du programme). Dans *test_formes.py*, on utilise `assert` pour tester une fonctionnalité qui n'est pas encore implémentée, l'exécution de ce fichier échouera tant que les méthodes de seront pas codées. À l'issue de cette partie, elle ne devra renvoyer plus aucune erreur !

Exercice 5 - Implémentez les méthodes d'affichage (`__str__`) de chacune des classes dans *formes.py*. Vous pourrez vérifier leur bon fonctionnement en exécutant *formes.py* (bouton `Run File – F5`), puis par exemple avec une commande `print(Rectangle(0, 0, 10, 10))` dans la console *IPython*.

Exercice 6 - Implémentez les méthodes d'accès (*getter/setter*) pour les champs privés de chacune des classes. Pour vérifier que les champs sont bien privés, le code suivant **doit** échouer avec une erreur `AttributeError` :

```
r = Rectangle(0, 0, 10, 10)
print(r.__x, r.__y, r.__w, r.__h)
```

Exercice 7 - Implémentez les méthodes `contient_point` des deux sous-classes. Vous vérifierez que les deux premiers `assert` des méthodes de test ne déclenchent pas d'erreur.

Exercice 8 - Implémentez les méthodes `redimension_par_points` de chacune des sous-classes. Le fichier *test_formes.py* doit à présent s'exécuter sans problème.

Solution du fichier *formes.py* :

```
class Forme:
    def __init__(self, x, y):
        self.__x = x
        self.__y = y

    def get_pos(self):
        return self.__x, self.__y

    def set_pos(self, x, y):
        self.__x = x
        self.__y = y
```

```
    def translation(self, dx, dy):
        self.__x += dx
        self.__y += dy

class Rectangle(Forme):
    def __init__(self, x, y, l, h):
        Forme.__init__(self, x, y)
        self.__l = l
        self.__h = h

    def __str__(self):
        return f"Rectangle d'origine {self.get_pos()} et de dimensions {self.__l}x{self.__h}"

    def get_dim(self):
        return self.__l, self.__h

    def set_dim(self, l, h):
        self.__l = l
        self.__h = h

    def contient_point(self, x, y):
        X, Y = self.get_pos()
        return X <= x <= X + self.__l and \
               Y <= y <= Y + self.__h

    def redimension_par_points(self, x0, y0, x1, y1):
        self.set_pos(min(x0, x1), min(y0, y1))
        self.__l = abs(x0 - x1)
        self.__h = abs(y0 - y1)

class Ellipse(Forme):
    def __init__(self, x, y, rx, ry):
        Forme.__init__(self, x, y)
        self.__rx = rx
        self.__ry = ry

    def __str__(self):
        return f"Ellipse de centre {self.get_pos()} et de rayons {self.__rx}x{self.__ry}"

    def get_dim(self):
        return self.__rx, self.__ry

    def set_dim(self, rx, ry):
        self.__rx = rx
        self.__ry = ry

    def contient_point(self, x, y):
        X, Y = self.get_pos()
        return ((x - X) / self.__rx) ** 2 + ((y - Y) / self.__ry) ** 2 <= 1

    def redimension_par_points(self, x0, y0, x1, y1):
        self.set_pos((x0 + x1) // 2, (y0 + y1) // 2)
        self.__rx = abs(x0 - x1) / 2
        self.__ry = abs(y0 - y1) / 2

class Cercle(Ellipse):
    def __init__(self, x, y, r):
        Ellipse.__init__(self, x, y, r, r)

    def __str__(self):
        return f"Cercle de centre {self.get_pos()} et de rayon {self.get_dim()}"

    def get_dim(self):
        return Ellipse.get_dim(self)[0]

    def set_dim(self, r):
        Ellipse.set_dim(self, r, r)

    def redimension_par_points(self, x0, y0, x1, y1):
        r = min(abs(x0 - x1), abs(y0 - y1)) / 2
        self.set_dim(r)
```

```
        self.set_pos(round(x0 + r if x1 > x0 else x0 - r),
                     round(y0 + r if y1 > y0 else y0 - r))
```

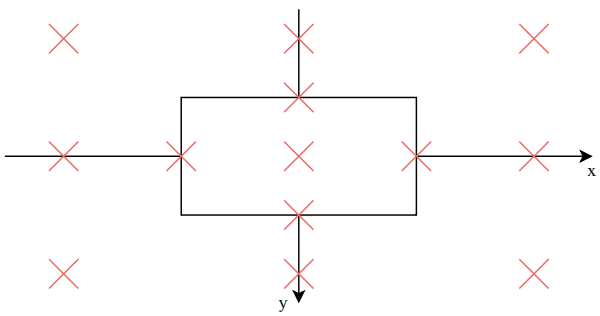
Tests unitaires (bonus)

Une fois développées, vos classes vont être utilisées pour des besoins que vous n'aviez pas forcément anticipés. Elles vont évoluer également pour acquérir de nouvelles fonctionnalités, et vont gagner en complexité. Dans ces conditions, il est courant de voir apparaître des bugs. Une pratique répandue pour améliorer la qualité logicielle est de définir des *tests unitaires*, c'est-à-dire de créer des situations extrêmes et vérifier que vos fonctions donnent toujours de bons résultats. Les tests unitaires serviront à documenter les cas d'utilisation supportés, et également à vous assurer qu'une modification de votre code n'a pas introduit un bug (une *régression*).

Voici une liste de tests relativement exhaustive pour la classe **Rectangle**

```
def test_Rectangle():
    r = Rectangle(-20, -10, 40, 20)
    assert r.contient_point(0, 0)
    assert r.contient_point(-20, 0)
    assert r.contient_point(0, -10)
    assert r.contient_point(20, 0)
    assert r.contient_point(0, 10)
    assert not r.contient_point(-40, 0)
    assert not r.contient_point(0, -20)
    assert not r.contient_point(40, 0)
    assert not r.contient_point(0, 20)
    assert not r.contient_point(-40, -20)
    assert not r.contient_point(40, -20)
    assert not r.contient_point(40, 20)
    assert not r.contient_point(-40, 20)
    reference = str(r)
    r.redimension_par_points(-20, 10, 20, -10)
    assert str(r) == reference
    r.redimension_par_points(20, 10, -20, -10)
    assert str(r) == reference
    r.redimension_par_points(20, -10, -20, 10)
    assert str(r) == reference
    r.redimension_par_points(-20, -10, 20, 10)
    assert str(r) == reference
```

Exercice 9 - Exécutez ce test sur votre code, et corrigez les éventuels bugs. Représentez ensuite, dans un logiciel de dessin (ex. diagrams.net), le rectangle et les positions des points qui sont testés. Quels bugs sont visés par chacun de ces tests ?



La rédaction de tests unitaires consiste souvent à anticiper les bugs courants, pour améliorer la qualité du logiciel dès sa conception. On cherche donc délibérément à provoquer des situations difficiles à gérer (ex. points *sur* le bord du rectangle). De telles situations sont par exemple :

- le choix de `<` ou `<=` dans le code;
- le traitement de valeurs négatives;
- les erreurs d'arrondis dans les opérations avec `float` ;
- la gestion de valeurs nulles (ex. largeur ou hauteur).

Exercice 10 - Dessinez une ellipse dans votre logiciel de dessin, et représentez tous les points qu'il convient de tester avec `contient_point`. Pour chaque point (ou groupe de points), indiquez le type de bug qu'il vise en particulier. Implémentez ces tests dans *test_formes.py*.

Exercice 11 - Dans le cas du cercle, la différence principale avec l'ellipse est la méthode `redimension_par_points` qui nécessite de placer le centre du cercle au plus près du premier point. Proposez des tests qui permettent de vérifier que la méthode positionne toujours correctement le cercle dans la boîte englobante d'entrée.