RAPPORT TD 3 Héritage - Polymorphisme - Liaison

Étudiant : Eliezer Mahugnon DJIHINTO, TD2 STI 3A, INSA CVL

Exercice 1 : Polymorphisme, liaison

a. Création des méthodes Draw() dans la classe FigureGeometrique

```
public void draw() {
    System.out.println("Drawing a geometrical shape");
}

public void draw(int zone) {
    System.out.println("Drawing the " + zone + "th zone of a shape");
}
```

b. Création de la méthode Draw() dans la classe Rectangle()

```
public void draw() {
    System.out.println("Drawing a rectangle");
}
```

c. Après exécution du code à la consigne précédente, on constate que ce code illustre deux formes de polymorphisme :

```
Rectangle r = new Rectangle();
r.draw(4);
r.draw();
```

- **le polymorphisme de surcharge** pour la méthode *draw(int zone)* : [le choix est fait à la compilation selon les paramètres donnés].

- **le polymorphisme d'héritage** pour la méthode *draw()* : [le choix est fait à l'exécution suivant le type réel de l'objet].

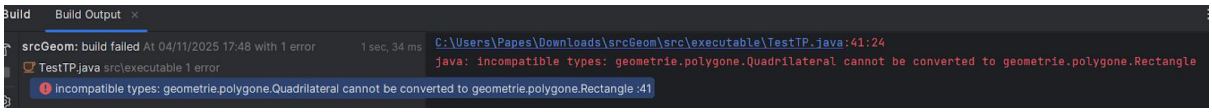
d. Malgré les différences de type déclaré de listFig et des objets auxquels les éléments du tableau font référence, le code donné fonctionne parce que toutes les classes (*Rectangle() ; Quadrilateral(); Ellipse()*) héritent de FigureGeometrique()

```
- public class Conical extends FigureGeometrique
- public class Ellipse extends Conical
- public class Polygone extends FigureGeometrique
public class Quadrilateral extends Polygone
- public class Rectangle extends Quadrilateral
```

Il s'agit du **transtypage implicite (upcasting)**.

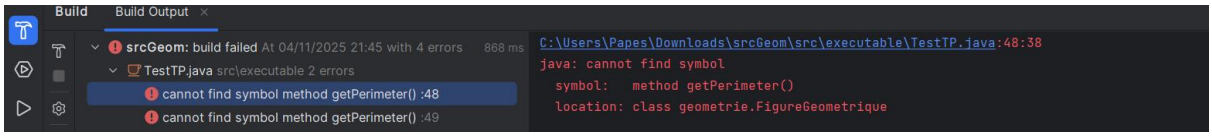
e. Code testé (erreur survenue : ***incompatibles types Quadrilateral cannot be converted to Rectangle***)

```
Rectangle rec = new Quadrilateral();
```



En Java, un objet d'une superclasse (Quadrilateral) ne peut pas être affecté à une variable d'une sous-classe (Rectangle) sans transtypage explicite, car tous les quadrilatères ne sont pas des rectangles. Ce type de transtypage (descendant / downcasting) est interdit par le compilateur.

f. Code testé (erreur survenue : **cannot find symbol method getPerimeter() in class FigureGeometrique**)



Dans FigureGeometrique(), la méthode getPerimeter() déclarée abstraite est mise en commentaire : Le compilateur ne sait donc pas que la méthode existe dans les sous-classes (notamment *Rectangle()*) pour les indexer.

Pour utiliser la méthode getPerimeter() de Rectangle sur listFig[0], il faut appliquer le transtypage explicite (downcasting) sur la dernière ligne :

```
((Rectangle) listFig[0]).getPerimeter();
```

Ce cast "convertit" temporairement la référence de type *FigureGeometrique()* en référence de type Rectangle, pour pouvoir accéder à ses méthodes spécifiques.

g. Non, ce n'est pas possible d'appeler directement (sans modifier l'implémentation des classes) la méthode getPerimeter() de Quadrilateral pour un objet dont le type réel est Rectangle.
En Java, la méthode appelée dépend du type réel de l'objet (ici Rectangle) et non du type déclaré ou du cast effectué.

h. La **liaison tardive** est le mécanisme par lequel Java choisit, au moment de l'exécution, la méthode correspondant au type réel de l'objet et non à son type déclaré.

Elle s'applique aux méthodes d'instance redéfinies (overriding) et permet le polymorphisme d'héritage : une même référence de type général peut exécuter des comportements différents selon la classe réelle de l'objet.

i. Déclaration et duplication de la variable d'instance code de type int et de visibilité public dans les classes.

j. Code testé

```
for (int i = 0; i <= 2; i++) {
        System.out.println(listFig[i].code);
```

On obtient :

```
0
0
0
```

Lorsqu'on affiche listFig[i].code, la valeur affichée est toujours celle de la classe FigureGeometrique (0), car les attributs ne sont pas soumis à la liaison dynamique : ils sont liés statiquement au type déclaré de la référence. Même si chaque objet réel a un code de valeur différente, l'accès se fait via la référence. Il s'agit là de la **liaison statique** : le **champ accédé dépend du type déclaré de la variable, pas du type réel de l'objet.**

Pour accéder aux valeurs spécifiques de code dans les sous-objets (Quadrilateral, Rectangle, Ellipse) à l'aide du transtypage (ici le transtypage explicite) :

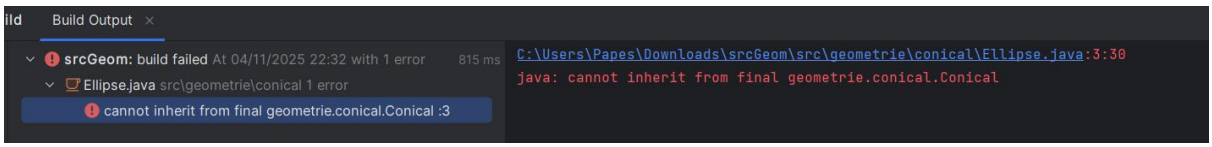
```
        System.out.println(((Rectangle) listFig[0]).code);       // 2
        System.out.println(((Quadrilateral) listFig[1]).code);   // 1
        System.out.println(((Ellipse) listFig[2]).code);         // 3
```

Et pour accéder à ces mêmes valeurs sans transtypage, on peut déclarer des variables de type spécifique aux objets pour accéder directement au champ correspondant :

```
        Rectangle r = new Rectangle();
        System.out.println(r.code);  // 2
        Quadrilateral q = new Quadrilateral();
        System.out.println(q.code);  // 1
        Ellipse e = new Ellipse();
        System.out.println(e.code);  // 3
```

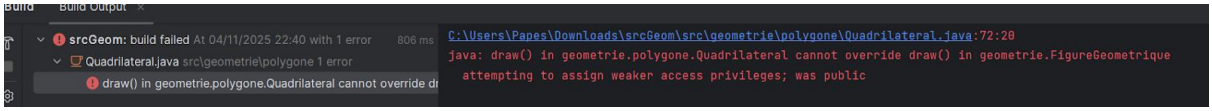
Exercice 2 : Visibilité, abstract, final

a. Ajout de ***final*** à la déclaration de la classe Conical() :



En ajoutant le mot-clé ***final*** à la déclaration de la classe Conical, on empêche toute autre classe d'en hériter. Or, dans le projet, la classe Ellipse étend déjà Conical, ce qui provoque l'erreur ci-dessus à la compilation.

b. Restriction de la visibilité de la méthode draw() dans Quadrilateral à protected.



En rendant la méthode draw() de Quadrilateral protected, on limite son accès aux sous-classes et aux classes du même package. Lorsqu'on appelle donc draw du Quadrilateral depuis le ***main*** (dans *TestTP*), le compilateur renvoie cette erreur car la méthode n'est plus visible au niveau de la classe ***TestTP***.
Pour corriger cela, il suffit de remettre la visibilité ***public*** à la place de *protected*, afin que la méthode soit accessible depuis le programme principal.

c. Ajout du code dans TestTP :

```
FigureGeometrique gs = new FigureGeometrique();
```

Erreur :

error: FigureGeometrique is abstract; cannot be instantiated

Cette instruction provoque donc l'erreur ci-dessus.
En effet, ***FigureGeometrique*** est une **classe abstraite** : elle définit une structure commune (attributs, signatures de méthodes) mais ne fournit pas d'implémentation complète.
On ne peut donc pas créer d'objet directement à partir d'une classe abstraite.