Exercice 1. La surcharge de méthodes et l'influence des droits d'accès

```
class A
{
   public void f(int n, float x) { … }
   public void f(float x, int n) { … }
   public void f(float x1, float x2) { … }
}
```

- Avec la classe `A` et les déclarations suivantes dans le main d'une classe `TestA`:
  `A a=new A(); short p; int n1, n2; float x;` , est-ce que les appels suivants sont corrects et si oui quelle méthode est appelée ?
  - a.f(n1, x) ;
  - a.f(x, n1) ;
  - a.f(p, x) ;
  - a.f(n1, n2) ;
  - a.f(p, p) ;
- Même exercice en supposant que `void f(int n, float x)` est `private`.
- Même exercice, toujours en supposant que `void f(int n, float x)` est `private`, mais cette fois en supposant en plus que les déclarations ont lieu dans une méthode de `A`.

Exercice 2. Les membres de classe (membres `static`)

- Réaliser une classe *Compteur* qui permet d'attribuer un numéro unique à chaque nouvel objet créé (1 au premier, 2 au deuxième etc.). On ne cherchera pas à réutiliser les numéros des objets éventuellement détruits. On dotera la classe uniquement d'un constructeur, d'une méthode *getId* fournissant le numéro de l'objet et d'une méthode *getIdMax* fournissant le numéro du dernier objet créé. On écrira un petit programme de test au sein d'une classe *TestCompteur*.
- Adapter la classe *Compteur* afin de lire au clavier le numéro initial des objets. On s'assurera que ce numéro est positif. On utilisera un bloc d'initialisation statique.

Exercice 3. Les objets membres et les packages

```
class Point2D
{ public Point2D(float x, float y) { this.x = x ; this.y = y ; }
  public void deplace(float dx, float dy) { x += dx ; y += dy ; }
  public void affiche() { System.out.println("coord = " + x + " " + y ) ; }
  private float x, y ;
}
```

- En complétant si nécessaire la classe précédente, proposer une classe `Segment2D` permettant de manipuler les segments d'un plan et disposant des méthodes suivantes :
  - `Segment2D (Point2D p1, Point2D p2)`
  - `float longueur()`
  - `void affiche()`
  - `void deplaceP1(float dxP1, float dyP1)`
  - `void deplaceP2(float dxP2, float dyP2)`
- Proposer un programme de test.
- La classe `Segment2D` réalise-t-elle une *composition* ou une *agrégation* ?

- Proposer une classe `Triangle2D` munie des méthodes *perimetre* et *aire*. Même chose pour une classe `Rectangle2D`.
- Mettre les classes `Point2D`, `Segment2D`, et `Triangle2D` – `Rectangle2D`, respectivement, dans les **packages** nommés `geometry.zeroDim`, `geometry.oneDim` et `geometry.twoDim`. Mettre les classes de Test (de la forme TestXXX) dans un **package** nommé `tests`.
- Est-ce que l'instruction `import geometry.*` est possible pour pouvoir utiliser toutes les classes des 3 packages ?

Exercice 4. L'emploi de this

```
class Point2D
{ public Point2D(float x, float y) { this.x = x ; this.y = y ; }
  public void deplace(float dx, float dy) { x += dx ; y += dy ; }
  public void affiche() { System.out.println("coord = " + x + " " + y ) ; }
  private float x, y ;
}
```

Compléter la classe `Point2D` de l'exercice précédent des méthodes suivantes :

- Une méthode d'instance `distance` qui retourne la distance entre le point courant et un autre point. On utilisera le mot clé `this` pour accéder aux membres du point courant.
- Un constructeur sans argument qui utilise le constructeur avec 2 arguments en initialisant les coordonnées à (0,0).
- Un constructeur avec 1 argument qui utilise le constructeur avec 2 arguments en initialisant les coordonnées à (abs,0).
- **Optionnel**: Une méthode `rotation(double angleEnRadians)` qui permet de faire subir une rotation aux coordonnées 2D (le centre de rotation est l'origine). On pourra passer en coordonnées polaires pour ajouter l'angle en question, puis revenir aux coordonnées cartésiennes. La classe `Math` contient des méthodes utiles (`cos`, `sin` etc.). On utilisera le mot clé `this` pour accéder aux membres du point courant.