# TP de PPO en JAVA

Polytech Lille GIS4 2018-2019

**Objectif :** généricité paramétrique.

Les sources (incomplètes) de la hiérarchie de classes génériques du cours (pages 26 et 27) :
`Liste<E>, ListeContigue<E>, ListeChainee<E>`
sont fournies dans le répertoire :
`/home/gisEns/bcarre/public/tpProgGenerique`

Copiez ce répertoire de travail sur votre compte.

## 1 Figures

A des fins de test, programmer une hiérarchie de classes de figures géométriques simplifiées :

- `Rectangle` : (copier le code du 1er TP) décrits par 2 points `origine` et `corner` (`java.awt.geom.Point2D.Double`), un constructeur :
  `Rectangle(double xo, double yo, double xc, double yc)`
  et les méthodes :
  - `double surface()`
  - `toString()` qui renvoie la chaîne de caractères :
    "`Rectangle: (<origine>, <corner>)`"
- `Cercle` : décrits par leur `centre` (`java.awt.geom.Point2D.Double`), leur `rayon` (`double`), un constructeur :
  `Cercle(double xcentre, double ycentre, double rayon)`
  et les méthodes :
  - `double surface()`
  - `toString()` qui renvoie la chaîne de caractères :
    "`Cercle: (<centre>, rayon)`"
- `Figure` : sur-classe abstraite des classes de figures qui factorise simplement la méthode abstraite '`double surface()`'.

## 2 Listes génériques

Compléter l'implantation de la hiérarchie de classes de listes génériques fournie dans le répertoire `tpProgGenerique`, notamment :

- pour l'implantation chaînée, ajouter la classe `Cellule<E>` de cellule de liste (indication : rendre générique la classe `Cellule` utilisée en cours (pages 7 à 9) pour l'implantation de `ListeChaineeRectangles`)
- programmer les méthodes `get(i)` et `primitiveGet(i)` en vous inspirant de `add(x)`. On provoquera l'exception `IndexOutOfBoundsException` si `i` n'est pas dans les bornes.

# 3 Application

Programmer une classe `Application` fournissant un `main`. Cette classe sera complétée et testée progressivement avec les méthodes (`static`) de manipulation de listes génériques demandées ci-dessous.

## 3.1 Polymorphisme d'inclusion entre classes génériques

1. Programmer une méthode :
   `void initRectangles(Liste<Rectangle> l)`
   qui crée (en dur) quelques rectangles et les range dans la liste `l`. Programmer de façon similaire une méthode `initCercles`.
2. Programmer une méthode (règle "joker" de la page 34 du poly) :
   `void printListe(l)`
   qui permet d'afficher les éléments d'une liste quelconque (de cercles ou de rectangles) avec leur indice de rangement.
3. Dans le `main`, déclarer des listes de rectangles et de cercles telles que :
   `Liste<Rectangle> lRectangles = ...`
   `Liste<Cercle> lCercles = ...`
   et les instancier en choisissant l'une des classes d'implémentation (chainée ou contigue). Tester ainsi le polymorphisme d'inclusion entre les sous-classes d'implémentation et la classe `Liste` (règle de la page 29 du poly) :
   - à la déclaration
   - en utilisant `initRectangles`, `initCercles` et `printListe` sur ces listes.

## 3.2 Généricité contrainte supérieurement par "extends" (règle de la page 42 du poly)

1. Programmer une méthode :
   `double cumulSurfaces(l)`
   qui calcule la somme des surfaces d'une liste de figures quelconques.
2. Programmer une méthode:
   `Figure maxSurface(l)`
   qui renvoie la figure de plus grande surface de la liste `l`.
3. Programmer une méthode:
   `void etat(l)`
   qui affiche la liste (`printListe(l)`), le résultat de `cumulSurfaces(l)` et de `maxSurface(l)`.
4. Dans le `main`, tester `etat(l)` sur les listes de rectangles et de cercles.

## 3.3 Généricité contrainte inférieurement par "super" (règle de la page 37 du poly)

1. Programmer une méthode :
   `void appendRects(Liste<? super Rectangle> dest, Liste<Rectangle> src)`
   qui permet de copier les éléments d'une liste `src` de rectangles dans toute liste `dest` pouvant contenir des rectangles.
2. Tester dans le `main` la généricité contrainte sur le paramètre `dest` (vérifier l'état des listes en utilisant `etat(l)`) :

   (a)
   ```
   // avec lr de type Liste<Rectangle>
   appendRects(lr, lRectangles);
   ```

(b)
```
// avec lFigures de type Liste<Figure>
appendRects(lFigures, lRectangles);
```

(c)
```
// avec lc de type Liste<Cercle>
appendRects(lc, lRectangles);
```

3. Faire de même avec des cercles :
   `void appendCercles(Liste<? super Cercle> dest, Liste<Cercle> src)`
4. Remarquer en particulier que ces deux méthodes peuvent s'appliquer à une liste de figures telle que `lFigures`. Ainsi, après 2.(b) et 3.(b) la liste `lFigures` contient bien des figures quelconques, rectangles ou cercles. Vérifier cela grâce à `etat(l)`.

## 3.4 Généralisation

1. En remarquant la similitude entre `appendRects` et `appendCercles`, factoriser leur code en programmant une méthode générique unique `'void append(dest,src)'` qui permet de copier les éléments d'une liste quelconque dans une autre.
2. Appliquer `append` à l'exemple des figures :
   (a) remplacer dans les tests 3.3.2 et 3.3.3 les appels à `appendRects` et `appendCercles` par des appels à `append`. Vous devez obtenir exactement les mêmes résultats tant à la compilation (notamment pour les cas qui ne doivent pas passer tels que 2.(c)) qu'à l'exécution.
   (b) vérifier que cette méthode s'applique également entre listes de figures, en testant par exemple :
   ```
   // avec lf de type Liste<Figure>
   append(lf, lFigures);
   etat(lf).
   ```
3. En reprenant la hiérarchie des classes de comptes bancaires, vérifier que la méthode générique `append` s'applique également sur des listes de différents types de `Compte`. Effectuer des tests tels que 3.3.2.(a)(b)(c) et 3.4.2(b).