Série F : Récursivité

"Loi de Hofstadter: Il faut toujours plus de temps que prévu, même en tenant compte de la loi de Hofstadter."

Table des matières

Exercice F.1: power

- Réalisez la méthode récursive **power** qui calcule X^N pour un réel X et un entier N positif ou zéro. Il n'est pas nécessaire de définir un traitement spécial pour les cas où X^N n'est pas défini.
- Complétez la méthode récursive **power** pour calculer X^N pour un réel X et un entier N positif, négatif ou zéro.

Exercice F.2: gcd

Réalisez la méthode récursive **gcd** qui calcule le plus grand commun diviseur de deux entiers x et y selon la définition suivante:

$$\begin{cases} \texttt{gcd(x,0) = x} \\ \texttt{gcd(x,y) = gcd(y, x \% y)} \qquad \text{pour } \mathbf{y}>0 \end{cases}$$

Exercice F.3: machin & truc

Réalisez cet exercice sur papier, sans le programmer !

Soit la méthode suivante :

```
public boolean machin(int x)
{
     if (x==0) return true;
     else      return !machin(x-1);
}
```

a) Indiquez la suite des appels récursifs et le résultat pour l'appel **machin(3)**.

b) A quoi sert la méthode **machin** ?

c) Pour quelles valeurs de X la méthode **machin** fournit-elle un résultat ?

Soit la méthode suivante :

```
public double truc( double x, int y)
{
    if      (y==0)       return 1;
    else if (machin(y)) return truc(x*x,y/2);
    else                 return truc(x,y-1) * x;
}
```

d) Indiquez la suite des appels récursifs de **truc** et le résultat pour l'appel **truc(2,5)** ?

e) A quoi sert la méthode **truc** ?

"Cette phrase est sans signification puisqu'elle se réfère à elle-même"

Exercice F.4: Fibonacci

L'origine des nombres de ***Fibonacci*** remonte à 1202 à Leonardo de Pisa, (fils de Bonacci – *"filius Bonacci"*) et repose sur un principe simple: évaluer à quelle vitesse des lapins peuvent se produire dans des circonstances idéales. Prenez un couple de lapins qui vient de naître, supposez que ces lapins peuvent procréer dès l'âge d'un mois et qu'il en résulte chaque fois un autre couple de lapins.

Au début du troisième mois, vous aurez deux couples de lapins. Au début du quatrième mois, le couple initial engendre un autre couple de lapins, ce qui donne un total de trois couples. Au cours du quatrième mois, le deuxième couple se met également à procréer, ce qui donne cinq couples au début du cinquième mois. Même si les lapins ont démarré lentement, ils rattrapent vite le temps perdu. Imaginez, au bout de 25 ans à ce rythme, il y aurait

222 232 244 629 420 445 529 739 893 461 909 967 206 666 939 096 499 764 990 979 600

couples de lapins!

Version moderne du même problème:

Imaginez que tous les utilisateurs d'un programme (*freeware* – bien sûr) agissent comme suit: Le premier jour, ils testent le programme et à partir du deuxième jour, ils passent chaque jour une copie du programme à un de leurs amis qui agit alors de la même façon ...

a) Réalisez la méthode récursive **`fibo_r`** qui calcule le nombre de couples de lapins (de copies du programme) après **`n`** mois (jours)!

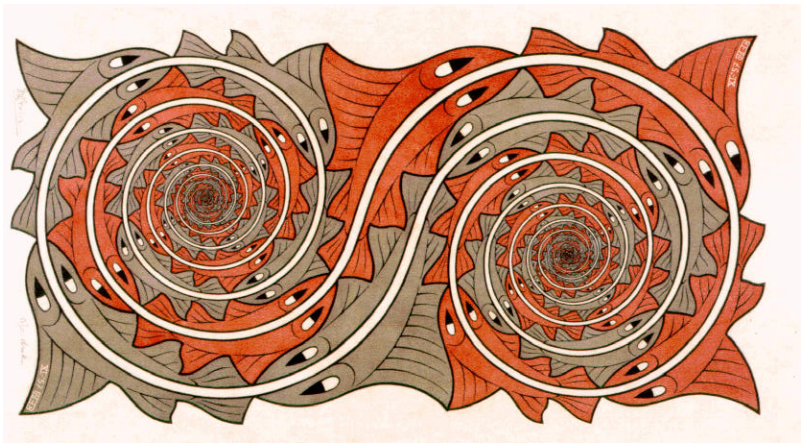
b) Réalisez la méthode itérative **`fibo_i`** qui calcule le nombre de couples de lapins (de copies du programme) après **`n`** mois (jours)!

c) Testez et comparez les deux méthodes **`fibo_r`** et **`fibo_i`**!

d) Représentez graphiquement (arbre) la suite des appels récursifs pour **`fibo_r(6)`**!

e) Déterminez le nombre d'appels récursifs nécessaires pour calculer **`fibo_r(1)`**, **`fibo_r(2)`**,...,**`fibo_r(6)`**!
Trouvez une formule générale pour déterminer le nombre d'appels récursifs qui sont générés au cours du calcul de **`fibo_r(n)`**!
Exprimez la formule sous forme d'une fonction **`nar(n)`** (nombre d'appels récursifs)!

f) Essayez de trouver une formule qui décrit la relation entre **`fibo_r(n)`** et **`nar(n)`**!



M. C. Escher

Exercice F.5: Suites et fonctions récursives

La suite de Newton permet de calculer la racine carrée d'un nombre **a** réel positif. Elle est définie par la relation suivante:

$$\begin{cases} u_0 = a \\ u_{n+1} = \frac{1}{2}\left(u_n + \frac{a}{u_n}\right) \end{cases}$$

Ecrire la méthode récursive **newton** qui retourne comme résultat le **n**ième terme de la suite de **newton** pour un réel **a** donné. **Limitez le temps de calcul** au strict nécessaire!

Exercice F.6: Séries de Taylor

Les développements en série de Taylor des fonctions *exp*, *cosinus* et *sinus* de la variable réelle *x* peuvent s'écrire :

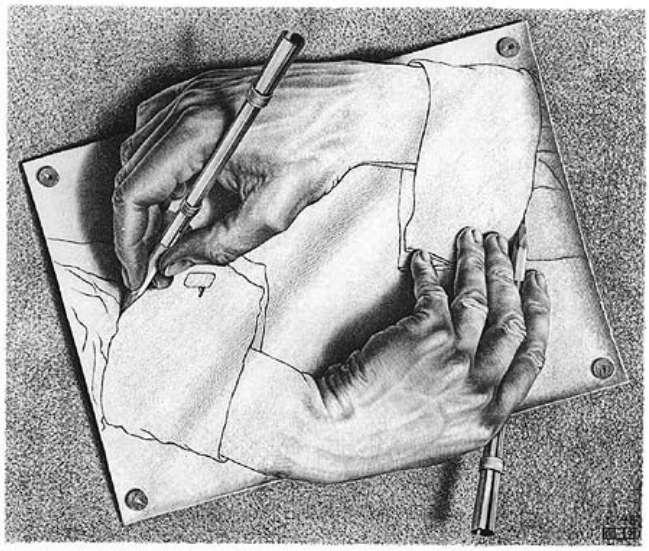
$$e^x = \frac{x^0}{0!} + \frac{x^1}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \ldots = \sum_{n=0}^{\infty} \frac{x^n}{n!}$$

$$\cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \ldots = \sum_{n=0}^{\infty} \left(\frac{x^{4n}}{(4n)!} - \frac{x^{4n+2}}{(4n+2)!} \right)$$

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots = \sum_{n=0}^{\infty} \left(\frac{x^{4n+1}}{(4n+1)!} - \frac{x^{4n+3}}{(4n+3)!} \right)$$

Pour chacune des séries ci-dessus, écrire une méthode récursive qui effectue le développement de la série jusqu'au terme ***n*** (pour un réel ***x*** et un entier positif ***n*** donnés).

"La phrase que j'écris maintenant est la phrase que vous lisez maintenant."



Exercice F.7: Précision d'approximation

a) Calculez la racine carrée à l'aide de la suite de newton, mais cette fois, indiquez la précision du résultat désiré au lieu du nombre d'itérations. P.ex. : pour obtenir la racine carrée de 2 avec une précision à 0.0000001 près, l'appel initial est:

newton(2, 0.0000001);

Définissez une méthode supplémentaire si nécessaire !

b) Calculez e^x selon la série de Taylor, mais en indiquant la précision de calcul désirée au lieu du nombre d'itérations. P.ex. : pour obtenir la racine carrée de e^5 avec une précision à 0.0000001 près, l'appel initial est :

exp(5, 0.0000001);

Définissez une méthode supplémentaire si nécessaire !

Exercice F.8: Boucles → récursivité

Pour les cas suivants, éliminez la boucle en effectuant les mêmes opérations à l'aide d'un sous-programme récursif! Indiquez aussi les parties qui font appel aux sous-programmes récursifs!

a)

```
for (int i=1 ; i<=10 ; i++)
      System.out.println(i);
```

En général:

```
for (int i=1 ; i<=n ; i++)
      <bloc d'instructions>
```

b)

```
for (int i=0 ; i<=3 ; i++)
   for (int j=0 ; j<=9 ; j++)
      System.out.println(i * 10 + j);
```

En général:

```
for (int i=0 ; i<=n ; i++)
   for (int j=0 ; j<=m ; j++)
      <bloc d'instructions>
```

Exercice F.9: Recherches linéaire et dichotomique

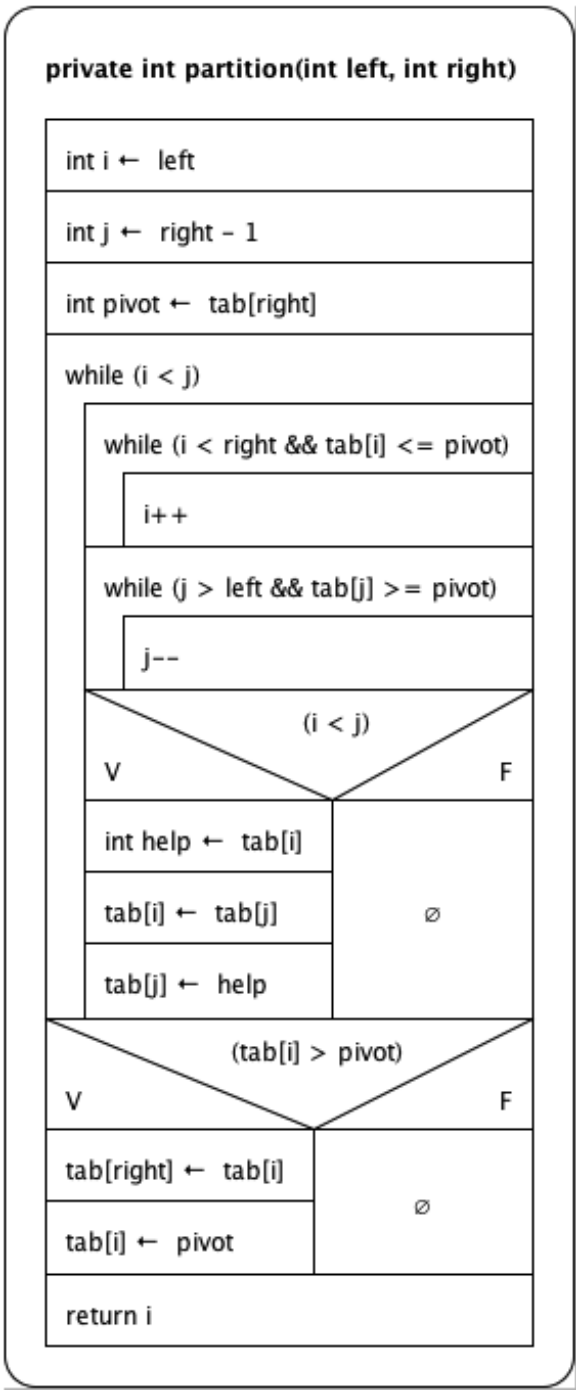
>>> Familiarisez-vous d'abord avec les tableaux statiques (**arrays**) <<<
>>> dans le chapitre 'Structures de données' <<<

1. Développez d'abord une classe qui contient une tableau statique **`tab`** qui peut être rempli de nombres entiers aléatoires. **`toString`** retourne la liste des nombres, **`printAll`** les affiche dans la fenêtre des messages.
2. Ajoutez une méthode (non récursive) **`selSort`** pour trier les nombres selon la méthode du tri par sélection.
3. Ajoutez une méthode récursive **`linearSearch`** qui recherche la position d'un nombre dans la liste en traversant le tableau élément par élément. La méthode retourne comme résultat la position du nombre dans la liste ou bien la valeur –1 si la valeur ne se trouve pas dans la liste. Déterminez le temps de recherche en employant **`System.currentTimeMillis()`**.
4. Développez une méthode récursive **`binarySearch`** qui recherche la position d'un nombre dans un ***tableau trié*** en utilisant le procédé de la **recherche dichotomique** ! La méthode retourne comme résultat la position du nombre dans la liste ou bien la valeur –1 si la valeur ne se trouve pas dans la liste. Déterminez le temps de recherche en employant **`System.currentTimeMillis()`**.
5. Comparez le temps de recherche des deux algorithmes pour des tableaux de différentes grandeurs et établissez un graphique.

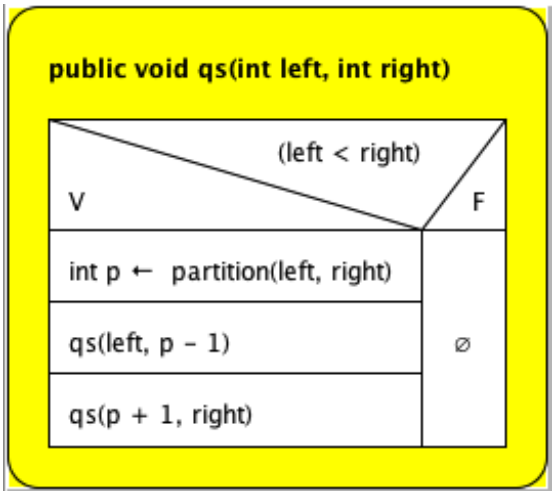
Exercice F.10: qs

Soit un tableau **tab** contenant des nombres entiers et soient deux nombres entiers **left** et **right** tels que : **0 ≤ left ≤ right ≤ tab.length-1**

1) En analysant le structogramme, déterminez l'effet de la méthode **partition** sur le contenu de **tab** et déterminez le résultat retourné !



Considérez ensuite la méthode récursive **qs** définie comme suit:



2) En analysant le structogramme, déterminez l'effet de la méthode **qs** sur le contenu d'une liste **tab** lors d'un appel par **qs(0, tab.length-1)**
3) Intégrez la méthode dans une classe contenant l'attribut **tab** pour tester la procédure **qs** (et **partition**).
4) Dans le même programme, réalisez la méthode non récursive **sl** qui a le même effet que **qs**, et qui a déjà été traitée auparavant.
5) Comparez la vitesse de travail des deux méthodes **qs** et **sl** pour des listes de différentes grandeurs!

Exercice F.11: Les tours de Hanoï

La légende des Tours de Hanoï remonte aux origines des temps :

"Dans le grand et fameux Temple de Benares, sous le dôme marquant le centre du monde, trône un socle de bronze sur lequel sont fixées trois aiguilles de diamant, chacune d'elle, haute d'une coudée et fine comme la taille d'une guêpe.

Sur une de ces aiguilles, à la Création, Dieu empila soixante quatre disques d'or pur, du plus grand au plus petit, le plus large reposant pur le socle de bronze.

Il s'agit de la tour de Bramah. Jour et nuit, inlassablement, les moines déplacent les disques d'une aiguille vers l'autre tout en respectant les immuables lois de Bramah qui obligent les moines à ne déplacer qu'un disque à la fois et à ne jamais le déposer sur un disque plus petit.

Quand les soixante quatre disques auront été déplacés de l'aiguille sur laquelle Dieu les déposa à la Création vers une des autres aiguilles, la tour, le temple, et les Brahmanes seront réduits en poussière, et le monde disparaîtra dans un grondement de tonnerre." [1]

En résumé: On a 3 aiguilles en face de soi, numérotées 1, 2 et 3 de la gauche vers la droite, et *n* disques de tailles toutes différentes entourant l'aiguille 1, formant un cône avec le plus gros en bas et le plus petit en haut. On veut amener tous les disques de l'aiguille 1 à l'aiguille 3 en ne prenant qu'un seul disque à la fois, et en s'arrangeant pour qu'à tout moment il n'y ait jamais un disque sous un disque plus gros. La légende dit que les moines passaient leur vie à Hanoi à résoudre ce problème pour *n=64*.

Ce problème compliqué à première vue peut être résolu à l'aide d'un algorithme récursif de quelques lignes seulement.

Familiarisez-vous avec le problème posé en essayant de le résoudre (sans ordinateur) pour un nombre limité de disques (p.ex. pour *n=4*). Essayez de généraliser votre méthode de résolution. Déduisez-en une méthode récursive ***hanoi*** dont l'en-tête est définie comme suit:

```
public void hanoi (int n, int i, int j)
```

avec: ***n*** nombre de disques à déplacer
i numéro de l'aiguille de départ
j numéro de l'aiguille d'arrivée

Exemple d'appel: **`hanoi(64,1,3)`** affiche tous les déplacements à faire pour déplacer 64 disques de l'aiguille 1 vers l'aiguille 3.

P.ex: la ligne '**`1->2`**' décrit le déplacement d'un disque de l'aiguille 1 vers l'aiguille 2.

Combien de déplacements faut-il faire pour résoudre le problème pour *n* disques? Démontrez votre hypothèse! En supposant que le déplacement d'un disque d'or prend une seconde, combien de temps mettront les moines pour provoquer ***la fin du monde*** ?

[1] *En réalité cette légende fut inventée par le mathématicien français E. Lucas en 1883*

Exercice F.12: Arbre fractal - *Fractal tree* - Baum-Fraktal

Dans une classe ***FractalTree***, développez une procédure ***drawVector*** qui sert à dessiner des segments (vecteurs) à l'écran (sur un canevas g). Paramètres de la méthode:

Graphics2D g	le canevas cible
Point2D p	les coordonnées du point de départ du segment (nombres réels)
double length **double angle**	la longueur du segment et son angle

Valeur retour de la méthode :

Point2D	les coordonnées de la fin (du dernier point) du segment

Point2D correspond à **Point**, mais mémorise les coordonnées en tant que réels (double), ce qui résulte en des calculs et des dessins beaucoup plus précis. L'instanciation se fait par `p=new Point2D.Double`. Les valeurs sont arrondies uniquement lors du dessin à l'écran. **Graphics2D** est dérivé de **Graphics**, mais offre plus de fonctionnalités.

Un arbre fractal binaire (fini et symétrique) peut être défini comme suit:

A partir d'une branche de longueur **L** et angle **A** partent deux arbres dont les premières branches ont la longueur **L*φ** *(φ<1)* et les angles **A+ΔA** et **A-ΔA** *(0<ΔA<180)*. Les branches d'une valeur minimale (p.ex. **L<1**) ne portent plus de bifurcations – elles sont considérées comme les *feuilles* de l'arbre.

Définir une procédure récursive ***drawFractalTree*** qui sait dessiner des arbres fractals. Choisissez les paramètres de façon appropriée et servez-vous de la procédure ***drawVector***.

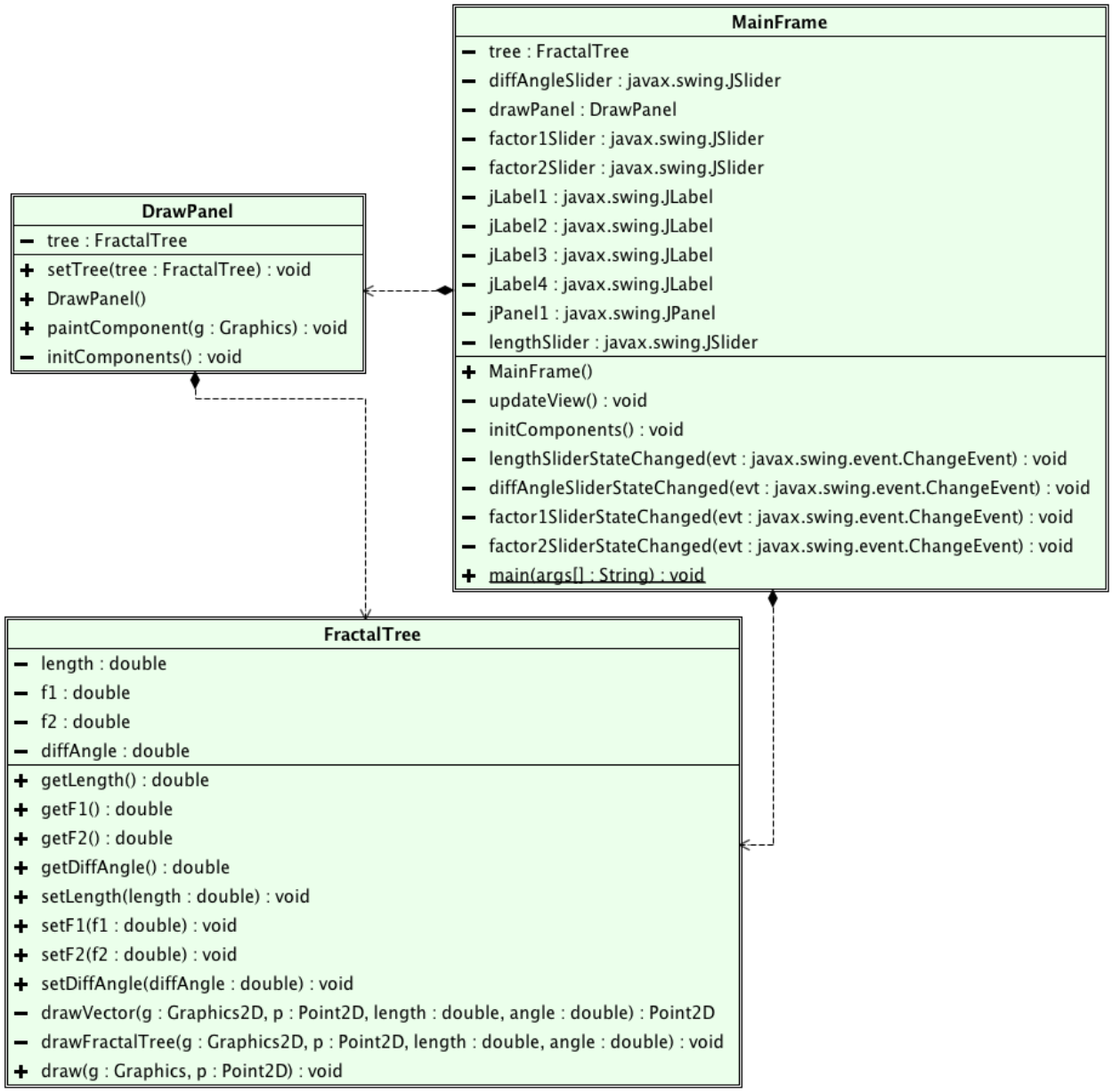
Définir une méthode **draw** dans **FractalTree** pour lancer la représentation (→ voir UML page suivante).

Améliorations:

- Vous obtenez des arbres asymétriques en utilisant deux valeurs différentes φ_1 et φ_2 pour les deux branches de l'arbre.
- Vous arrivez à produire de vrais petits chefs d'œuvres artistiques si vous utilisez des couleurs différentes pour des branches de différentes longueurs.
- En plus, il est possible tracer les branches avec des épaisseurs proportionnelles à leur longueur (p.ex: L/10), ce qui donne aux arbres un aspect plus 'naturel'.

Expérimentez avec différentes valeurs, par exemple:

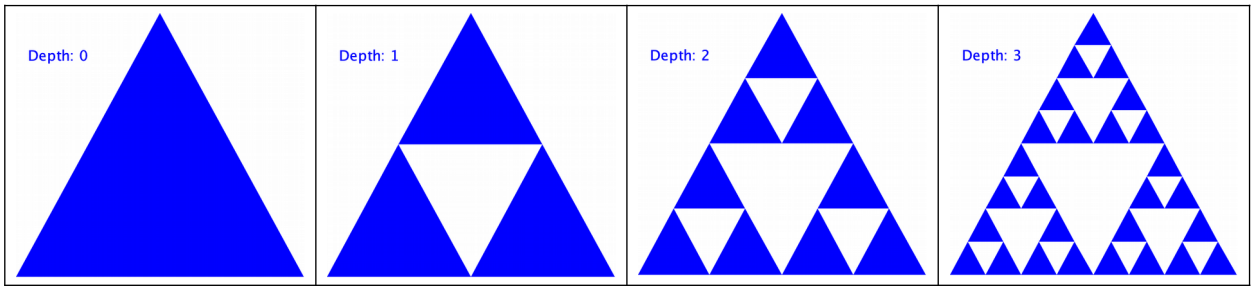
Longueur initiale	φ_1	φ_2	ΔA
300	0,6	0,6	90,45,20,10
250	0,7	0,7	90,45,20,10
200	0,75	0,75	45
200	0,65	0,8	45
200	0,65	0,85	25
200	0,85	0,1 ... 0,7	50
150	0,8	0,85	25



Quels parallélismes détectez-vous entre la structure d'un arbre fractal et les procédés utilisés dans les exercices précédents?

Exercice F.13: Fractale: Triangle de Sierpinski

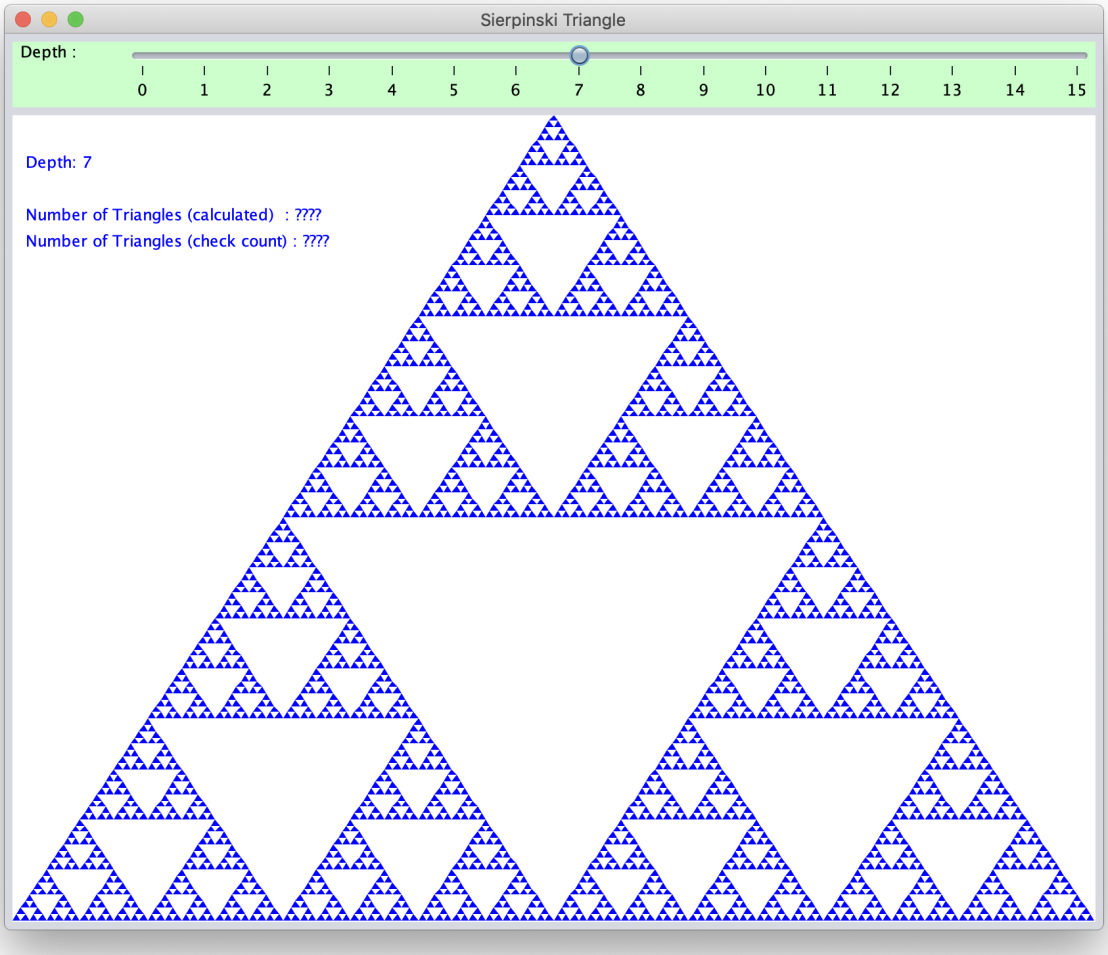
Dans une classe ***SierpinskiTriangle***, développez une méthode ***draw*** qui dessine la fractale connue sous le nom de "Triangle de Sierpinski". Pour les profondeurs de récursion de zéro à 3, le triangle de Sierpinski se présente comme suit:



Utilisez la méthode **fillPolygon** pour dessiner les triangles.

Déduisez, calculez et affichez le nombre de triangles qui sont dessinés pour les différentes profondeurs de récursion. Vérifiez en faisant compter les triangles dessinés.

Quelle modification faut-il faire pour que le programme dessine les triangles jusqu'au degré de récursion où les triangles sont encore tout juste visibles.



Exercice F.14: File Scanner

Créez le projet **FileScanner** et ajoutez la classe **FileScanner** qui doit posséder une méthode publique statique :

ArrayList<File> scanDirectory(File file).

Cette méthode publique est utilisée pour retourner dans une liste tous fichiers et sous-dossiers qui se trouvent dans le dossier donné comme paramètre ou récursivement dans la structure de sous-dossiers qui se trouve dans ce dossier.

Créez une interface graphique pour le projet.

Indications :

- Utilisez les méthodes de la classe **java.io.File**, notamment: **isDirectory**, **isFile**, **listFiles**. Consultez *JavaDoc* à ce sujet.
- L'instruction **jfc.setFileSelectionMode(JFileChooser.DIRECTORIES_ONLY);** permet de limiter le choix du dialogue à des dossiers.