UPMC SORBONNE UNIVERSITÉS

UFR d'Ingénierie

21 octobre 2015

Licence d'Informatique (2015-2016)

Programmation Concurrente (3I001)

TD/TME

Thèmes 6 à 9

Thème 6

Objectifs

— Respiration

Exercices TD

Exercice 19 – Gestion d'un pont à voie unique

Nous considérons un système représentant le passage de véhicules sur un point à voie unique. Ce système comporte les composants suivants :

— un **pont** à voie unique, gardé par des **feux**,
— des **véhicules** divisés en deux catégories selon leur sens de circulation (Nord-Sud ou Sud-Nord),
— un **contrôleur** qui gère les feux.

Nous considèrerons que :

— lorsqu'une voiture arrive devant un feu rouge, elle attend que ce dernier passe au vert ;
— le contrôleur n'a aucune interaction directe avec les voitures. Il gère le changement de couleur des feux, qu'il réalise régulièrement après un délai aléatoire.

Question 1

Quelles vérifications doit effectuer le contrôleur pour éviter les collisions lorsqu'il décide d'inverser le sens de passage sur le pont ?

Question 2

Quelles sont les classes qui doivent implémenter l'interface `Runnable` ?

Question 3

Donnez le code de la classe `Pont`.

Question 4

Pour représenter la couleur d'un feu, nous utiliserons un type énuméré. Donnez la déclaration de ce type et le code de la classe `Feu`.

Les véhicules sont tous identiques, distingués par le sens dans lequel ils circulent (`NORD_SUD` ou `SUD_NORD`, défini par un type énuméré) et un identifiant, unique pour un sens de circulation donné.

Le comportement d'un véhicule est d'arriver au feu et d'attendre si nécessaire qu'il passe au vert, d'entrer sur le pont, de le traverser, d'en sortir et de se terminer. La traversée du pont sera matérialisée par une méthode exécutant une attente d'une durée aléatoire.

Question 5

Écrivez le type énuméré et la classe `Vehicule`.

Question 6

Le contrôleur doit périodiquement modifier les couleurs des feux pour permettre une circulation alternée, en évitant les risques de collision. La durée pendant laquelle un feu reste vert est donnée par une valeur aléatoire choisie entre deux bornes prédéfinies.

Donnez le code de la classe `Controleur`.

Question 7
Écrivez une classe `TestPont` permettant de tester les interactions entre les différents composants de ce système.

Question 8
Pour des raisons de sécurité, la décision est prise de limiter le nombre de véhicules présents simultanément sur le pont à une valeur constante `NB_MAX`. Donnez les modifications du système nécessaires pour la réalisation de cette nouvelle stratégie.

Question 9
Que se passe-t-il si une voiture s'arrête après avoir passé le feu et avant d'entrer sur le pont ? Peut-on mettre cette situation en évidence ?

Question 10
Comment peut-on résoudre ce problème ?

Exercices TME

Lors de chaque TME, vous devrez créer un répertoire pour chaque exercice, y mettre les fichiers s'y rapportant et soumettre ce répertoire à la fin du TME.

Exercice 20 – Gestion d'un garage à locomotives

On considère un ensemble de hangars composé de N unités pouvant chacune accueillir une locomotive. Le segment d'accueil permet d'entrer et de sortir des locomotives de la zone de stockage. Une locomotive entrante est dirigée vers un hangar donné (ou vers le segment d'accueil si elle est sortante) au moyen d'un segment de voie tournant. L'architecture du système est représentée dans la Figure 1.

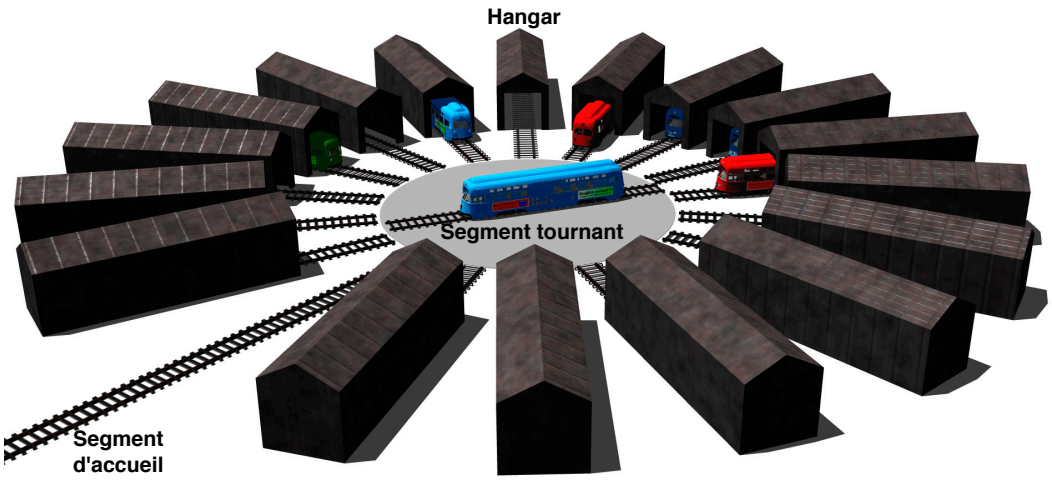

FIGURE 1 – Architecture du système pour $N = 17$

On souhaite représenter au moyen d'une application concurrente le pool de hangars ainsi que les locomotives qui s'y garent. Pour cela, on considèrera un système composé des éléments suivants :
- des locomotives,
- des hangars regroupés au sein d'un pool de hangars,
- le segment tournant.
- le segment d'accueil.

Pour accéder à un hangar, une locomotive commence par réserver le segment d'accueil. Elle appelle ensuite le segment tournant qui se positionne devant le segment d'accueil : la locomotive peut alors entrer dans le segment tournant et libérer le segment d'accueil. Elle attend ensuite que le segment tournant se positionne devant un hangar disponible (il y en a toujours un) pour sortir du segment et entrer dans le hangar.

On peut donc considérer dans notre système une classe `Loco` dont la méthode `run` est donnée ci-dessous :

```
public void run() {
    try {
        sAccueil.reserver();
        sTournant.appeler(0);
        sTournant.attendrePositionOK();
        sTournant.entrer(id);
        sAccueil.liberer();
        sTournant.attendrePositionOK();
        pHangars.getHangar( sTournant.getPosition() ).entrer(id);
        sTournant.sortir(id);
    }
    catch (InterruptedException e) {
    }
}
```

Le paramètre de la méthode `appeler` correspond à la position à laquelle doit se rendre le segment tournant (`0` est la position face au segment d'accueil) et un identifiant de locomotive est transmis à certaines méthodes pour permettre des affichages explicites.

La méthode `run` de la classe `SegTournant` est donnée ci-dessous :

```
public void run() {
    try {
        while (true) {
            attendreAppel();
            seDeplacer();
            attendreEntree();
            seDeplacer();
            attendreVide();
        }
    }
    catch (InterruptedException e) {
        return;
    }
}
```

Le but de cet exercice est de programmer ces méthodes (et d'autres...) de manière à garantir une progression synchronisée du segment tournant et d'une locomotive.

Question 1

Identifiez les conditions de blocage du segment tournant (i.e., les situations dans lesquelles le segment tournant doit attendre une action d'une locomotive pour pouvoir progesser).

Question 2

Faites de même pour une locomotive.

Nous avons fait le choix d'implémenter l'ensemble des méthodes de synchronisation dans la classe `SegTournant`.

Question 3

Identifiez les paires de méthodes correspondant à des actions synchronisées, les variables booléennes et les outils de synchronisation nécessaires pour implémenter les blocages identifiés précédemment.

Question 4
Donnez l'implémentation de la classe `SegAccueil` dont une instance représente le segment d'accueil.

Question 5
L'ensemble des hangars est regroupé au sein d'un pool de hangars. Ce pool doit offrir (entre autres) une méthode permettant de rechercher un hangar libre à l'intérieur du pool et renvoyant sa position dans le pool.

Donnez l'implémentation de la classe `Hangar` et de la classe `PoolHangars`.

Question 6
Écrivez le code de la classe `SegTournant`. On simulera le déplacement du segment par une attente.

À quel moment la variable indiquant que le segment a été appelé est-elle remise à faux ?

Question 7
Complétez la classe `Loco` et écrivez un programme de test permettant de vérifier le fonctionnement du système.

Thème 7

Objectifs

— Interface `ReadWriteLock`

Exercices TD

Exercice 21 – Le pont, encore...

Nous reprenons le système représentant le passage de véhicules sur un point à voie unique. La solution que nous avons construite n'est pas complètement satisfaisante, dans la mesure où elle introduit un lien entre la classe `Feu` et la classe `Pont` (on pourrait pour cela créer une sous-classe `FeuDePont` pour garder la généralité de `Feu`) et surtout, parce qu'elle appelle une méthode synchronisée à l'intérieur d'une méthode synchronisée, ce qui est une source potentielle d'interblocage.

Question 1
Comment garantir que le contrôleur ne peut pas inverser le feu alors qu'un véhicule a passé le feu (vert), mais n'est pas encore entré sur le pont ?

Question 2
Quelles sont les classes qui sont affectées par la mise en place de cette nouvelle stratégie ?

Question 3
Que faut-il modifier dans la classe `Vehicule` ?

Question 4
Proposez une nouvelle implémentation de la classe `Feu`.

Question 5
Peut-on rapprocher ce problème d'un problème de lecteur/écrivain ?

Question 6
Modifiez l'implémentation en conséquence.

Question 7
Quel problème peut poser cette solution ? Comment peut-on le résoudre ?

Exercices TME

Lors de chaque TME, vous devrez créer un répertoire pour chaque exercice, y mettre les fichiers s'y rapportant et soumettre ce répertoire à la fin du TME.

Exercice 22 – Optimisation des accès en lecture/modification

Nous souhaitons mettre en place un système dans lequel 3 classes de processus, des Producteurs, des Lecteurs et des Effaceurs accèdent à un ensemble de données. Cet ensemble n'est pas de taille fixe et est représenté en mémoire par une liste (une instance de la classe `ArrayList`). Nous nous placerons dans un cas simple où les données manipulées sont des entiers.

— les producteurs ajoutent des données, toujours à la fin de la liste ;
— les lecteurs parcourent la liste et affichent les données lues ;
— les effaceurs recherchent une valeur dans la liste et la suppriment s'ils la trouvent. Ils affichent un message d'erreur si la valeur n'est pas présente.

Dans un premier temps, *nous ne nous préoccupons pas des problèmes de synchronisation*.

Question 1
Écrivez une classe `EnsembleDonnees` offrant aux 3 classes de processus les méthodes nécessaires à la réalisation de leurs opérations. La méthode de suppression doit lever une exception (que vous définirez) lorsque la donnée n'est pas présente.

Vous n'utiliserez pas d'itérateur dans l'implémentation.

Question 2
Écrivez le code correspondant à chacune des classes de processus, ainsi qu'un programme de test qui crée 2 instances de chaque classe.

Lancez l'exécution. Que se passe-t-il ?

On souhaite maintenant synchroniser les accès à l'ensemble de données, en optimisant le parallélisme. Les accès en lecture peuvent se faire en parallèle alors qu'une modification de la liste est exclusive de tout autre accès.

Question 3
Proposez une modification de l'implémentation de la classe `EnsembleDonnees` qui réalise ces contraintes. Que constate-t-on à l'exécution ?

Question 4
Comment peut-on résoudre ce problème ?

Thème 8

Objectifs

— Vivacité, famine, interblocage.
— Interface `Lock : tryLock()`

Exercices TD

Exercice 23 – Prise de ressources multiples : le problème de la piscine

Le problème de la piscine est un schéma classique d'accès à des ressources multiples. La piscine dispose d'un certain nombre de cabines et de paniers. Pour pouvoir accéder au bassin, un baigneur entre dans une cabine, puis demande un panier dans lequel il dépose ses affaires. Il confie ce panier au personnel du vestiaire et quitte la cabine pour aller se baigner. Lorsqu'il a suffisamment nagé, le baigneur doit à nouveau accéder à une cabine. Il demande alors son panier, qu'il vide avant de le rendre. Il peut ensuite sortir de la cabine et quitter la piscine.

Question 1
Les ressources sont indifférenciées : toutes les cabines (resp. tous les paniers) sont identiques. Nous allons donc représenter un *ensemble* de ressources par une classe incluant un attribut qui donne le nombre d'exemplaires disponibles.

Écrivez une classe `EnsembleRessources` à partir de laquelle vous construirez une sous-classe pour les paniers et une sous-classe pour les cabines.

Question 2
La durée de la baignade est simulée par une attente de durée aléatoire. Écrivez une classe `Nageur` qui permette de représenter chaque nageur par un thread.

Question 3
Écrivez un programme de test pour un système avec 10 nageurs, 3 cabines et 5 paniers. Exécutez ce programme jusqu'à mettre en évidence un interblocage (vous pouvez éventuellement agir sur la durée de la baignade). Expliquez dans quelles circonstances se produit cet interblocage.

Question 4
Proposez une solution pour éviter cet interblocage sans modifier le nombre de ressources.

Question 5
Quel problème peut se poser si le nombre de nageurs n'est pas limité *a priori* ?

Question 6
On souhaite maintenant gérer l'ensemble des ressources comme une liste d'éléments (identiques). Comment peut-on garantir la cohérence de cette structure ?

Question 7
Proposez une nouvelle implémentation du problème de la piscine où les ressources sont individualisées et la cohérence assurée.

Exercice 24 – Dîner des philosophes

Nous souhaitons programmer un problème classique de programmation concurrente, “le dîner des philosophes”, dans lequel `N` philosophes sont assis autour d'une table ronde. Il y a `N` baguettes sur la table, chaque baguette est posée entre deux philosophes. Pour un dîner de 5 philosophes, la configuration est la suivante :

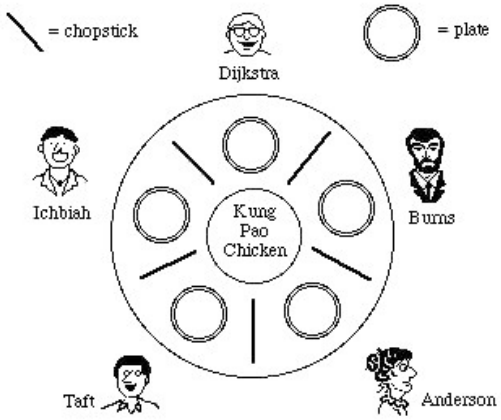

FIGURE 2 – Le dîner des 5 philosophes

Lorsqu'il est fatigué de penser, un philosophe décide de manger. Pour cela, il doit saisir la baguette qui se trouve à sa gauche et la baguette qui se trouve à sa droite. S'il y parvient, il peut manger et lorsqu'il est rassasié, le philosophe repose ses baguettes et repart dans une phase de réflexion.

Nous décidons dans un premier temps de programmer le partage des baguettes au moyen d'un objet synchronisé. Si une baguette n'est pas disponible, la méthode qui permet de prendre la baguette doit mettre le `thread` appelant en attente.

Question 1
Programmez la classe `ChopStick` qui décrit le comportement d'une baguette.

Question 2
Programmez la classe `Philo` qui décrit le comportement d'un philosophe. Vous imposerez un délai d'attente aléatoire entre la prise de baguette à gauche et la prise de baguette à droite.

Question 3
Programmez une classe `TestPhilo` qui permet de tester les deux classes que vous venez de programmer. Que constatez-vous ?

Nous souhaitons modifier la manière dont un philosophe s'empare de ses baguettes : s'il n'arrive pas à entrer en possession des deux baguettes, il repose celle qu'il a éventuellement réussi à prendre, se donne un délai d'attente et refait une tentative.

Question 4
Quel est le mécanisme de synchronisation adapté pour implémenter ce comportement ?

Question 5
Donnez une nouvelle version des classes `ChopStick` et `Philo`, et vérifiez que tous les philosophes parviennent à manger.

Testez à nouveau le comportement du système lorsqu'un philosophe se comporte de manière cyclique. Vous pouvez par exemple regarder le comportement du système lorsque chaque philosophe mange quatre fois.

Exercices TME

Lors de chaque TME, vous devrez créer un répertoire pour chaque exercice, y mettre les fichiers s'y rapportant et soumettre ce répertoire à la fin du TME.

Exercice 25 – De l'utilité d'un serveur multi-threads

Dans un système client/serveur, un ensemble de processus, appelés clients, envoient des requêtes à un processus serveur qui les traite et renvoie à chaque client la réponse correspondant à sa requête. Nous voulons programmer un système de ce type dans lequel :

— le serveur traite une seule requête à la fois : le traitement de chaque requête est exécuté directement dans le thread Serveur.

— un client ne peut envoyer une nouvelle requête qu'après avoir reçu la réponse à sa requête précédente.

Chaque client envoie au serveur une suite de requêtes numérotées. La réponse envoyée par le serveur lorsqu'il a traité une requête inclut une référence au client émetteur (certes inutile pour l'instant), le numéro de la requête et une valeur entière aléatoire représentant le résultat attendu par le client. Chaque réponse du serveur est une instance d'une classe `ReponseRequete`.

Question 1

Proposez une définition pour la classe `ReponseRequete`. Cette définition devra inclure une méthode `toString`.

Question 2

Programmez le thread principal qui crée un thread serveur et `NB_CLIENTS` threads clients. Comment faire pour terminer proprement l'application lorsque toutes les requêtes de tous les clients ont été servies ?

Un client émet vers le serveur des requêtes qui contiennent une référence vers le client émetteur (lui-même), un numéro de requête et un entier permettant de déterminer le type de traitement demandé. Nous considérons ici deux types de requêtes : les requêtes de type 1 émises par des clients dont l'identifiant n'est pas un multiple de 3, et les requêtes de type 2 émises par les autres clients. La soumission d'une requête est effectuée en appelant la méthode `soumettre` de la classe `Serveur`.

Un client qui a émis une requête attend de recevoir la réponse du serveur, effectue localement des opérations (dont l'exéuction est représentée par une durée aléatoire) avant de soumettre une nouvelle requête.

Question 3

Donnez le code de la classe `Client` lorsque chaque client envoie successivement 5 requêtes au serveur. Le serveur transmet au client la réponse à une requête en appelant la méthode **`void`** `requeteServie(ReponseRequete r)` de la classe `Client`. Vous devrez vous assurer qu'un client ne se termine pas avant d'avoir reçu une réponse à toutes ses requêtes.

Nous nous intéressons maintenant aux opérations du serveur. Celui-ci étant exécuté par un thread séparé, il doit comporter une méthode `run` dont le code est donné ci-dessous :

```
    public void run() {
        try {
            while (true) {
                attendreRequete();
                traiterRequete();
            }
        }
        catch (InterruptedException e) {
            System.out.println("Serveur interrompu");
        }
    }
}
```

Question 4

Donnez le code des méthodes `soumettre` (appelée par un client qui dépose une requête) et `attendreRequete` de la

classe `Serveur`. Le traitement de la requête étant exécuté dans le thread `Serveur`, la méthode `soumettre` appelée par le client signale simplement au serveur l'arrivée d'une nouvelle requête.

Vous devrez réfléchir aux conditions dans lesquelles un client ou le serveur se trouve bloqué en attente d'une action d'un autre processus.

La durée des opérations de traitement d'une requête de type 1 est représentée par un délai aléatoire borné, alors que nous représentons le traitement des requêtes de type 2 par une boucle infinie (pour simuler le cas d'un traitement très long).

Question 5

Complétez le code de la classe `Serveur` en ajoutant les variables d'instance nécessaires, le code du constructeur et le code de la méthode `traiterRequete`.

Testez l'ensemble du système de tâches. Celui-ci se termine-t-il correctement ?

Pour résoudre le problème de famine créé par l'exécution sur le serveur d'une requête très longue, nous modifions le fonctionnement de celui-ci : lorsqu'il reçoit une requête d'un client, le serveur crée un nouveau thread dédié à l'exécution du traitement de la tâche demandée.

Question 6

Quelles sont les modifications à apporter au système programmé précédemment ? Le problème de famine existe-t-il encore ?

Thème 9

Objectifs

— Pool de threads, interface `Executor`

Exercices TD

Exercice 26 – Utilisation d'un *Executor* pour le produit matriciel

L'opération de création et de destruction d'un thread est coûteuse. Avoir un thread par tâche n'est pas forcément une bonne stratégie, en particulier, il est généralement inutile d'avoir plus de threads que de processeurs pour les exécuter. Il est donc souvent intéressant de dissocier la création d'un ensemble de threads des tâches qu'ils vont exécuter.

Dans ce cas, une tâche est "chargée" sur un thread et lorsque cette tâche est terminée, le même thread, s'il n'a pas été détruit, peut être utilisé pour exécuter une autre tâche.

Nous allons appliquer cette approche au calcul du produit matriciel que nous avons réalisé avec un thread par tâche.

Question 1
Donnez l'instruction permettant de créer un ensemble (pool) de threads dont le nombre `NB_THREADS` ne varie pas au cours de l'exécution du programme.

Question 2
Nous avons construit une classe `CalculElem` qui implémente l'interface `Runnable` et dont le constructeur :

```
public CalculElem (MatriceEntiere m, MatriceEntiere m1, int i, MatriceEntiere m2, int j)
```

permet d'affecter une valeur à l'élément en position `(i, j)` dans la matrice `m`, résultat du produit de `m1` par `m2`. Donnez la suite d'instructions permettant de calculer l'ensemble des éléments de la matrice résultat en répartissant les calculs sur les `NB_THREADS` créés.

Question 3
Comment peut-on s'assurer de la terminaison du pool de threads et garantir que l'affichage de la matrice n'a pas lieu avant que l'ensemble des éléments aient été calculés ?

L'une des limites de l'interface `Runnable` est que la méthode `run` ne renvoie pas de résultat et ne peut pas lancer d'exception. Ces restrictions sont levées par la méthode `call` de l'interface `Callable`.

Question 4
Nous avons défini dans la classe `MatriceEntiere` une méthode

```
public static int produitLigneColonne (MatriceEntiere m1, int i, MatriceEntiere m2, int j)
```

qui peut lancer une exception si les données sont de tailles incompatibles. Écrivez le code permettant de créer un objet implémentant l'interface `Callable` et dont la méthode `call` retourne le résultat calculé par `produitLigneColonne`.

Question 5
Quelle méthode permet de soumettre la tâche pour exécution ? Comment peut-on récupérer le résultat ?

Question 6
Proposez une implémentation du produit matriciel multi-threads en utilisant des objets `Callable`.

Question 7
Quel problème pose l'utilisation d'un *CompletionService* pour récupérer les résultats des différents calculs ? Proposez une solution.

Exercices TME

Lors de chaque TME, vous devrez créer un répertoire pour chaque exercice, y mettre les fichiers s'y rapportant et soumettre ce répertoire à la fin du TME.

Exercice 27 – Visualiser l'effet d'un pool de threads : la courbe de Von Koch

La courbe de Von Koch est une fractale dont la construction repose sur le principe récursif suivant : partant d'un segment limité par deux points M et N, on divise ce segment en trois parties égales. Soient A et C les points créés par cette division, on construit entre ces deux points un point B tel que A, B, C est un triangle équilatéral. Puis on répète l'opération sur chacun des segments ainsi créés. Le principe est illustré par la Figure 3.

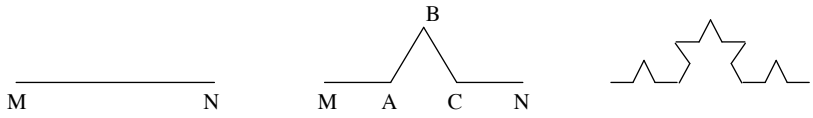

FIGURE 3 – Découpage d'un segment

La condition d'arrêt de la récursivité est liée à la taille des segments créés : lorsque la longueur de ceux-ci devient inférieure à une taille déterminée, on arrête le découpage. L'ensemble de classes ci-dessous propose une implémentation de ce tracé. La figure initiale est un triangle, et chaque côté du triangle subit une transformation.

```
import graphique.Fenetre;
import java.awt.Point;

public class Segment {
    private Fenetre f;
    private final Point m, n;

    public Segment (Fenetre f, Point m, Point n) {
        this.f = f;
        this.m = m;
        this.n = n;
    }

    public static double longueur(Point m, Point n) {
        return Math.sqrt( Math.pow (m.y - n.y, 2.0) + Math.pow (m.x - n.x, 2.0) );
    }

    public static void tracer (Fenetre f, Point m, Point n) {
        f.tracerLignePointAPoint(m, n);
    }
}

import graphique.Fenetre;
import java.awt.Point;

public class Cote {
    private final static double LG_MIN = 8.0;
    private final Fenetre f;
    private final Point m, n;

    public Cote (Fenetre f, Point m, Point n) {
        this.f = f;
        this.m = m;
        this.n = n;
    }
```

```
    public void tracer () {
        final double xa, ya, xb, yb, xc, yc;

        if ( Segment.longueur(m, n) > LG_MIN ) {
            xa = (2 * m.x + n.x) / 3.0;
            xc = (m.x + 2 * n.x) / 3.0;
            ya = (2 * m.y + n.y) / 3.0;
            yc = (m.y + 2 * n.y) / 3.0;
            xb = xa + ( xc - xa - (Math.sqrt(3.0) * (yc - ya)) ) / 2.0;
            yb = ya + ( yc - ya + (Math.sqrt(3.0) * (xc - xa)) ) / 2.0;

            Point a = new Point();
            a.setLocation(xa, ya);
            Point b = new Point();
            b.setLocation(xb, yb);
            Point c = new Point();
            c.setLocation(xc, yc);

            new Cote(f, m, a).tracer();
            new Cote(f, a, b).tracer();
            new Cote(f, b, c).tracer();
            new Cote(f, c, n).tracer();
        }
        else {
            Segment.tracer(f, m, n);
        }
    }
}

import graphique.Fenetre;
import java.awt.Point;

public class VonKochMono {
    private final static double LG_MIN = 8.0;
    Fenetre f;

    public VonKochMono (Fenetre f, Point a, Point b, Point c) {
        this.f = f;
        new Cote(f, b, a).tracer();
        new Cote(f, a, c).tracer();
        new Cote(f, c, b).tracer();
    }
}

import graphique.Fenetre;
import java.awt.Point;

public class TestVonKoch {

    public static void main (String[] args) {
        final int XMAX = 400;
        final int YMAX = 400;

        final Fenetre f = new Fenetre(XMAX, YMAX, "von Koch");

        Point v = new Point();
        v.setLocation(XMAX/2, YMAX/8);

        Point u = new Point();
        u.setLocation(v.x - (3.0 * XMAX/10.0), v.y + (0.5196 * XMAX));

        Point w = new Point();
        w.setLocation(v.x + (3.0 * XMAX/10.0), u.y);

        VonKochMono vk = new VonKochMono(f, u, v, w);

    }
}
```

Question 1

Compilez le programme précédent et exécutez le.

Question 2
Proposez une version multi-threads de ce programme dans laquelle chaque ligne affichée dans la fenêtre graphique est tracée par un thread différent.

Dans cette version, le nombre de threads créés par le programme est lié au nombre de lignes tracées. Nous voulons modifier le programme pour maîtriser le nombre de threads utilisés. On ne s'intéresse dans un premier temps qu'aux threads créés par le traitement d'un côté du triangle initial (et pas aux trois threads qui lancent chacun le traitement d'un côté).

Question 3
Proposez une nouvelle implémentation multi-threads de la classe `Cote`, en gérant le lancement des threads au moyen d'un objet de la classe `Executors`. Regardez ce qu'il se passe à l'exécution lorsqu'on autorise un seul thread actif par côté, puis avec des pools de threads de tailles différentes.

Question 4
Modifiez l'implémentation de manière à gérer maintenant un unique pool de threads pour l'ensemble du tracé.

Exercice 28 – Serveur utilisant un pool de threads

Nous reprenons l'exemple de l'application Client-Serveur, et nous voulons maintenant gérer l'exécution du traitement des requêtes au moyen d'un pool de threads. Comme ces requêtes produisent des résultats, nous allons utiliser un `ExecutorCompletionService` (ECS) pour pouvoir récupérer le résultat de chaque traitement.

L'une des conséquences de la création d'un ECS est que maintenant, contrairement à l'implémentation multi-threads où chaque tâche envoyait directement sa réponse au client, toutes les réponses aux requêtes passent par le serveur.

Le serveur doit donc, d'une part, attendre les requêtes en provenance des clients, d'autre part attendre les réponses en provenance des threads de traitement.

Question 1
Proposez une solution pour que la prise en compte de l'arrivée d'une réponse ne soit pas retardée lorsque le serveur est bloqué en attente d'une requête.

Question 2
Reprenez l'implémentation du Client-Serveur multi-threads pour que les requêtes soient maintenant traitées au moyen d'un pool de threads.

Question 3
Modifiez le programme pour qu'il se termine proprement lorsqu'il n'y a plus de requêtes à traiter.

Question 4
Quel est l'inconvénient d'utiliser un nombre fixe de threads pour répondre aux demandes des clients ? Peut-on y remédier, sans pour autant créer un thread par requête ?