La programmation concurrente

Exercice 1 : Simulateur de voitures

On veut étendre la classe *Voiture* afin que la méthode *rouler()* puisse d'exécuter de manière autonome, à savoir dans un thread dédié. Cette évolution correspond à un cas classique de développement dans lequel l'implémentation d'une classe change (passage à une implémentation mono-thread à multi-thread), sans les clients de la classe (ici la méthode *main()*) n'ai connaissance de ce changement. C'est une illustration de la notion de service et d'encapsulation.

Dans notre cas, il est demandé à la méthode *main()* de:

- Créer une ou plusieurs voitures et les placer dans une collection;
- Faire rouler chacune des voitures crées;
- Attendre 10 secondes;
- Arrêter les voitures.

Une voiture quant à elle roule en avançant d'un kilomètre toutes les secondes. Si elle n'a plus de carburant ou si on l'arrête, elle ne roule plus.

1. Implémenter l'ensemble de ces changements et tester leur bon fonctionnement. On rappelle que la classe *Voiture* hérite de la classe abstraite *Vehicule*.

```
public abstract class Vehicule implements Pilotable  {

    public static final int ACCELERATION = 10;

    public final int capaciteReservoir = 50;
    public final double consommation = 0.1;

    protected double essence;
    protected boolean roule;
    protected Moteur moteur;

    public class Moteur {
      private int kilometres;
      private int vitesse;

      public Moteur (int kilometres) {
          this.kilometres = kilometres;
          this.vitesse = 0;
      }

      public int getKilometres() {
          return kilometres;
      }

      public void ajouterKilometres(int kilometres) {
          this.kilometres += kilometres;
      }

      public int getVitesse() {
          return vitesse;
      }

      public void augmenterVitesse(int vitesse) {
          this.vitesse += vitesse;
      }

      public void diminuerVitesse(int vitesse) {
```

```
            this.vitesse -= vitesse;
        }
    }

    public Vehicule (int kilometres) {
        this.essence = capaciteReservoir;
        this.roule = false;
        moteur = new Moteur(kilometres);
    }

    public Vehicule () {
        this.essence = capaciteReservoir;
        this.roule = false;
        moteur = new Moteur(0);
    }

    public void accelerer() {
        if (this.roule == true) {
            moteur.augmenterVitesse(Vehicule.ACCELERATION);
        }
    }

    public void ralentir() {
        if (this.roule == true) {
            if (moteur.getVitesse() > 0) {
                moteur.diminuerVitesse(Vehicule.ACCELERATION);
            }

            if (moteur.getVitesse() == 0) {
                this.stopper();
            }
        }
    }

    public abstract void rouler();

    public void stopper() {
        this.roule = false;
    }

    public String toString() {
        StringBuffer sb = new StringBuffer ("Le vehicule ");
        if (this.roule == false) {
            sb.append("est ‡ l'arret. ");
        } else {
            sb.append("roule ‡ la vitesse de ");
            sb.append(moteur.getVitesse());
            sb.append("km/h. ");
        }
        sb.append("Il a parcouru ");
        sb.append(moteur.getKilometres());
        sb.append("kms. ");
        sb.append("Il reste ");
        sb.append(this.essence);
        sb.append(" litres d'essences dans son rÈservoir.");
        return sb.toString();
    }
}
```

```
import java.util.HashSet;
import java.util.Iterator;
```

```
public class Voiture extends Vehicule implements Orientable, Runnable {

    public static final int TEMPORISATION = 1000;

    private int orientation;

    private Thread threadRouler;

    public Voiture(int kilometres) {
      super(kilometres);
    }

    public Voiture() {
      super();
      orientation = 0;
    }

    public void tournerAGauche() {
      orientation = (orientation + 1) % Orientations.length;
    }

    public void tournerADroite() {
      orientation = (orientation + Orientations.length - 1) %
Orientations.length;
    }

    public void rouler() {
      this.roule = true;
      threadRouler = new Thread(this);
      threadRouler.start();
    }

    public void run() {

      System.out.println("Le vehicule dÈmarre...");
      while (this.roule && this.essence > 0) {
          moteur.ajouterKilometres(moteur.getVitesse());
          this.essence -= this.consommation * moteur.getVitesse();
          try {
            Thread.sleep(Voiture.TEMPORISATION);
          } catch (InterruptedException e) {
            e.printStackTrace();
          }
          System.out.println(this);
      }
      System.out.println("Le vÈhicule est arretÈ.");

    }

    public static void main(String[] args) {

      int nbVoitures = 2;
      int dureeRoulage = 10;

      HashSet<Voiture> voitures = new HashSet<Voiture>();

      for (int i = 1; i>= nbVoitures; i++) {
          voitures.add(new Voiture());
      }

      System.out.println("Les voitures crées :");
      System.out.println(voitures);
```

```
        System.out.println("Les voitures démarrent...");
        Iterator<Voiture> it = voitures.iterator();
        while (it.hasNext()) {
            Voiture v = it.next();
            v.rouler();
        }

        for (int i = 0; i < dureeRoulage; i++) {
            try {
              Thread.sleep(Voiture.TEMPORISATION);
            } catch (InterruptedException e) {
              e.printStackTrace();
            }
            System.out.println(voitures);
        }

        System.out.println("Les voitures s'arretent...");
        it = voitures.iterator();
        while (it.hasNext()) {
            Voiture v = it.next();
            v.stopper();
        }

        System.out.println(voitures);
    }
}
```

2. Lorsqu'un véhicule est crée et roule, combien de threads sont actifs dans la JVM ?

Il y a deux thread, un pour la méthode *rouler()* et un autre pour la méthode *main()*. Montrer aux étudiants comment on peut nommer un thread lors de sa construction et récupérer les noms des threads par *Thread.getCurrentThread().getName()*.

Exercice 2 : Le modèle Producteur/Consommateur

Le modèle producteur/consommateur est un modèle standard de la programmation concurrente. Il correspond au cas où certains objets produisent des ressources et d'autres les consomment. L'exemple que nous prenons ici est celui d'une boite aux lettres dans laquelle un producteur peut déposer un message. Plusieurs consommateurs sont en attente de messages. Celui-ci sera lu

1. Ecrire une classe *BoiteAuxLettres* (BAL) sachant que :
   - La boite ne peut contenir qu'un seul message (Chaine de caractères) ;
   - Elle possède une méthode pour déposer un message ;
   - Elle possède une méthode pour lire un message.

```
import java.util.HashSet;
import java.util.Iterator;

public class BoiteAuxLettres {

    private String message;

    public BoiteAuxLettres () {
      message = null;
    }

    public synchronized String lireMessage(String consommateur) throws
InterruptedException {
      while (message == null) {
          System.out.println(consommateur + " en attente...");
```

```
            this.wait();
        }
        String messageLu = message;
        message = null;
        System.out.println("Message lu :" + consommateur + ":" + messageLu);
        this.notifyAll();

        return messageLu;
    }

    public synchronized void deposerMessage(String producteur, String
nouveauMessage) throws InterruptedException {

        while (message != null) {
            System.out.println(producteur + " en attente...");
            this.wait();
        }

        message = nouveauMessage;
        System.out.println("Message déposé :" + producteur + ":" + message);
        this.notifyAll();
    }

    public static void main(String args[]) {

        int nbProducteurs = 2;
        int nbConsommateurs = 1;

        BoiteAuxLettres bal = new BoiteAuxLettres();

        HashSet<Producteur> producteurs = new HashSet<Producteur>();
        HashSet<Consommateur> consommateurs = new HashSet<Consommateur>();

        for (int i = 1; i <= nbProducteurs; i++) {
            producteurs.add(new Producteur("P" + i, bal));
        }

        for (int i = 1; i <= nbConsommateurs; i++) {
            consommateurs.add(new Consommateur("C" + i, bal));
        }

        Iterator<Producteur> itProducteurs = producteurs.iterator();
        while (itProducteurs.hasNext()) {
            Producteur p = itProducteurs.next();
            p.produireLettres();
        }

        Iterator<Consommateur> itConsommateurs = consommateurs.iterator();
        while (itConsommateurs.hasNext()) {
            Consommateur c = itConsommateurs.next();
            c.consommerLettres();
        }

        try {
            Thread.sleep(10000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        itProducteurs = producteurs.iterator();
        while (itProducteurs.hasNext()) {
            Producteur p = itProducteurs.next();
            p.arreterProduire();
```

```
        }

        itConsommateurs = consommateurs.iterator();
        while (itConsommateurs.hasNext()) {
            Consommateur c = itConsommateurs.next();
            c.arreterConsommer();
        }
    }
}
```

2. Ecrire une classe *Producteur* sachant que :
   - Un producteur dépose à intervalles réguliers des messages dans la BAL ;
   - Si la boite contient déjà un message, le producteur est mis en attente.

```
public class Producteur implements Runnable {

    private static long Temporisation = 1000;

    private String nom;
    private BoiteAuxLettres bal;
    private int nombreMessages;
    private boolean producteurActif;
    private Thread thread;

    public Producteur (String nom, BoiteAuxLettres bal) {
      this.nombreMessages = 0;
      this.nom = nom;
      this.bal = bal;
      this.producteurActif = false;
    }

    public void produireLettres() {
      thread = new Thread(this);
      this.producteurActif = true;
      thread.start();
    }

    public void arreterProduire() {
      this.producteurActif = false;
    }

    public void run() {

      try {

          Thread.sleep((long) (Producteur.Temporisation * Math.random()));

          while (this.producteurActif == true) {
            String message = new String("Message " + nombreMessages + "
from " + nom);
            nombreMessages++;

            bal.deposerMessage(nom, message);
            Thread.sleep(Producteur.Temporisation);
          }

      } catch (InterruptedException e) {
          e.printStackTrace();
      }
    }
}
```

3. Ecrire une classe *Consommateur* sachant que :
   - Le consommateur lit à intervalles réguliers des messages dans la BAL ;
   - Si la boite ne contient pas de message, il est mis en attente.

```
public class Consommateur implements Runnable{

    private static long Temporisation = 1000;

    private String nom;
    private BoiteAuxLettres bal;
    private int nombreMessages;
    private boolean consommateurActif;
    private Thread thread;

    public Consommateur (String nom, BoiteAuxLettres bal) {
      this.nombreMessages = 0;
      this.nom = nom;
      this.bal = bal;
      this.consommateurActif = false;
    }

    public void consommerLettres() {
      thread = new Thread(this);
      this.consommateurActif = true;
      thread.start();
    }

    public void arreterConsommer() {
      this.consommateurActif = false;
    }

    public void run() {

      try {

          Thread.sleep((long) (Consommateur.Temporisation *
Math.random()));

          while (this.consommateurActif == true) {
            String message = bal.lireMessage(nom);
            nombreMessages++;
            Thread.sleep(Consommateur.Temporisation);
          }
      } catch (InterruptedException e) {
          e.printStackTrace();
      }

    }
}
```

4. Tester votre modèle avec plusieurs producteurs et consommateurs. Pour ce faire, placer les producteurs et consommateurs dans des collections et utiliser des itérations pour démarrer et arrêter les threads associés.

Voir question 1.