VIVET Florian 19/01/22

Système : TP2 - Moniteurs

1) Simulation d'une pâtisserie

A)

On a deux patrons : producteur/consommateur et lecteur/rédacteur.
Le patron lecteur/rédacteur permet de gérer l'accès de rédacteurs et lecteurs a un fichier ou une base de donnée : Le rédacteur accède en écriture et le lecteur en lecture.
Cependant, ici dans le cas d'une pâtisserie, on souhaite que les clients puissent acheter des gâteaux (et donc les retirer de la pâtisserie). Comme les lecteurs n'accèdent à la ressource qu'en lecture, il ne sont pas censés la modifier, ce patron n'est donc pas adapté ici.

B)

Ici on a une liste de Gâteaux alimentée par des Pâtissiers (les producteurs) et consommée par des Clients (les consommateurs). On doit gérer le fonctionnement du programme pour qu'il n'y ai qu'un seul producteur ou consommateur qui accède à une ressource de la liste à un instant donné.
On aura donc deux type de *threads* : Clients et Pâtissiers à synchroniser sur une ressource : la liste de Gâteaux.

Pour surveiller l'accès à la liste des gâteaux, on aura donc une Pâtisserie qui servira de moniteur.

On a donc une classe Pâtisserie qui gère l'accès aux Gâteaux avec les méthodes *dépose()* et *achète()*.
Les fonctions *dépose()* et *achète()* sont *synchronized*, ce qui assure l'exclusivité sur la ressource durant l'exécution du code de la fonction.

```
public class Patisserie {

    public ArrayList<Gateau> stock;

    public Patisserie() {
        this.stock = new ArrayList<>();
    }

    public synchronized void depose(Gateau gateau){
        stock.add(gateau);
        this.notify();
    }

    public synchronized Gateau achete(){
        while(stock.isEmpty()){
            try {
                this.wait();
            } catch (InterruptedException ex) {
                Logger.getLogger(Patisserie.class.getName()).log(Level.SEVERE, null, ex);
            }
        }
        Gateau gateau = this.stock.remove(0);
        this.notify();
        return gateau;
    }

    public ArrayList<Gateau> getStock() {
        return stock;
    }
}
```

Les *threads* sont lancés depuis une classe main en instanciant un *thread* qui prend un client ou un pâtissier en paramètre. Ensuite on lance les *threads* avec la méthode *start()* depuis la même classe

```java
public class Main {

    public static void main(String[] args){
        Patisserie patisserie = new Patisserie();
        for (int i = 0; i <= 3; i++) {
            Thread t1 = new Thread(new Client("client "+i, patisserie));
            Thread t2 = new Thread(new Patissier("patissier "+i, patisserie));
            t1.start();
            t2.start();
        }
    }

}
```

Les Clients et les Pâtissiers sont instanciés en connaissant le même objet Pâtisserie et appelleront respectivement les méthodes *achète()* et *dépose()*. Dans un premier temps, il effectueront leur tache un nombre limité de fois (ici 10) ;

```java
public class Client implements Runnable{

    public String nom;
    public Patisserie patisserie;

    public Client(String nom, Patisserie patisserie) {
        this.nom = nom;
        this.patisserie = patisserie;
    }

    @Override
    public void run() {
        int i = 0;
        while(i < 10){
            Gateau gateau = this.patisserie.achete();
            System.out.println(this.nom + "a acheté un gateau");
            i++;
        }
    }
}
```

```java
public class Patissier implements Runnable{

    public String nom;
    public Patisserie patisserie;

    public Patissier(String nom, Patisserie patisserie) {
        this.nom = nom;
        this.patisserie = patisserie;
    }

    @Override
    public void run() {
        int i = 0;
        while(i < 10) {
            this.patisserie.depose(new Gateau());
            System.out.println(this.nom + "a déposé un gateau");
            i++;
        }
    }
}
```

2) Simulation d'une pâtisserie avec collection ThreadSafe

On utilise ici une ArrayBlockingQueue à la place d'une ArrayList de Gâteaux. L'ArrayBlockingQueue permet de gérer l'ajout et la récupération d'éléments de la queue en s'occupant de faire attendre ou de notifier les threads. Ainsi, c'est la queue qui jouera le rôle de moniteur.

Afin d'ajouter des éléments à la queue, on peut utiliser la fonction *offer()*, qui ajoute un élément si il est possible de le faire sans dépasser la taille maximun de le queue.

Pour prendre un élément de la liste, on peut utiliser la fonction *take()*, qui récupère l'élément en tête de file, ou met le thread en attente jusqu'à ce qu'un élément soit disponible

Ainsi j'ai repris la classe Patisserie pour l'adapter à cette nouvelle BlockingQueue :

```
public class Patisserie {

    public ArrayBlockingQueue<Gateau> stock;

    public Patisserie() {
        this.stock = new ArrayBlockingQueue(10);
    }

    public void depose(Gateau gateau){
        stock.offer(gateau);
    }

    public Gateau achete(){
        Gateau gateau = null;
        try {
            gateau = this.stock.take();
        } catch (InterruptedException ex) {
            Logger.getLogger(Patisserie.class.getName()).log(Level.SEVERE, null, ex);
        }
        return gateau;
    }

    public ArrayBlockingQueue<Gateau> getStock() {
        return stock;
    }
}
```

3) Fin de programme

A) Pour stopper les producteurs quand la file est pleine, on peut utiliser le code de retour de la fonction offer : elle renvoie false quand la file est pleine. On modifie alors la fonction dépose() et la fonction run de Patissier

dépose() récupère le retour de offer et renvoie un booléen au patissier :

```
public boolean depose(Gateau gateau){
    return stock.offer(gateau);
}
```

dépose() renvoie *true* si l'ajout a été fait et *false* si la file est pleine. Dans ce cas on arrête le thread :

```
@Override
public void run() {
    int i = 0;
    boolean continuer = true;
    while(continuer) {
        continuer = this.patisserie.depose(new Gateau());
        System.out.println(this.nom + " a déposé un gateau");
        i++;
    }
}
```

B)

C)