IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

# TD5 – Le tampon borné

## Première concurrence en Java

### Fondements théoriques du développement des logiciels concurrents

## Objectifs

À la fin de l'activité, les élèves devront être capables de :

- expliquer le besoin de concurrence dans un système logiciel ;
- expliquer les notions de *threads*, tâche, exclusion mutuelle, section critique et ressource partagée ;
- définir et d'activer des *threads* en Java selon l'approche par composition ;
- identifier le besoin d'exclusion mutuelle et de le mettre en œuvre en utilisant un verrou.

## Le problème du tampon borné

Dans ce problème classique en informatique concurrente, deux types d'*entités*, les producteurs et les consommateurs, partagent un tampon de taille fixe. Les producteurs remplissent le tampon avec des données. Simultanément, les consommateurs retirent des données du tampon pour les traiter. Par exemple, un clavier produit des caractères qui sont consommés par le logiciel qui gère l'affichage à l'écran.

Des contraintes s'appliquent à un tel système. Les producteurs ne doivent pas ajouter de données lorsque le tampon est plein et les consommateurs ne doivent pas récupérer des données d'un tampon vide.

### Exercice 1 (*Un modèle non concurrent*)

Une conception objet de la solution au problème du tampon borné se trouve dans la figure 1. Le tampon borné est représenté par un objet de type `Stock`. Nous supposerons que les consommateurs consomment les produits dans un ordre LIFO (« *Last In First Out* »).

▷ **En supposant que vous avez une réalisation Java de ces classes, donnez le code d'une méthode `main` d'une classe `Main`, permettant de créer un producteur qui produit**

**deux produits puis un consommateur qui consomme ces produits. Donnez la trace d'exécution (l'ordre d'exécution) de ce code. Peut-on obtenir une trace différente selon l'exécution ?**

## Exercice 2 (*Des processus concurrents*)

Nous allons maintenant modifier l'application pour créer un système concurrent composé de producteurs et de consommateurs qui ont une *« vie »* indépendante. Ainsi, leurs productions et consommations d'un stock (c'est-à-dire les appels aux méthodes `produce` et `consume`) peuvent se faire dans un ordre indéterminé.

▷ **Question 2.1 :**
**Quelle pourrait être la tâche (dans le sens donné dans le cours) d'un producteur et d'un consommateur ? Donnez le code des méthodes `produce` de la classe `Producer` et `consume` de la classe `Consumer`.**

Pour pouvoir exécuter de manière indépendante la tâche des producteurs et des consommateurs, il faut associer à chaque tâche un fil d'activités ou *thread*. En Java, une tâche est définie dans une méthode `run` d'un objet de type `Runnable`. Un fil d'activités est un objet de type `Thread`.

▷ **Question 2.2 :**
**Donnez le code Java pour définir la tâche des consommateurs (et les producteurs).**

▷ **Question 2.3 :**
**Donnez le code Java pour associer la tâche des producteurs et des consommateurs à un *thread* et pour l'activer. Quelle est la trace d'exécution ?**

## Exercice 3 (*L'accès exclusif au tampon*)

Dans la solution précédente, le producteur et le consommateur accèdent à une *« ressource partagée »* : le tampon. De plus, ces accès se font de manière indépendante : un producteur peut donc être en train d'exécuter la méthode `produce` (qui utilise la méthode `add` du `Stock`), lorsque le consommateur exécute sa méthode `consume` (qui utilise la méthode `remove` du `Stock`).

▷ **Question 3.1 :**
**Si le code des méthodes `add` et `remove` de la classe `Stock` est celui donné ci-dessous, la trace suivante est-elle possible :**
***ligne 1 à ligne 6 de `add("tomate")` ; ligne 1 à 2 de `add("carotte")` ; ligne 1 à 8 de `remove()` ; ligne 3 à 6 de `add("carotte")` ?***
**Pourquoi n'est-elle pas « logique » ?**

```
1  public void add(Product p) {
2    if (p == null)
3      return;
4    if (!isFull())
5      this.content[this.size++] = p;
6  }
```

```
1  public Product remove() {
2    Product p = null;
```

```
3     if (!isEmpty()) {
4       p = this.content[--this.size];
5       this.content[this.size] = null;
6     }
7     return p;
8   }
```

Pour éviter le problème identifié dans la question précédente, le bloc des opérations de manipulation du tampon (la ressource partagée) doit être accédée de manière exclusive par un *thread* : si un *thread* est en train d'exécuter de telles opérations, aucun autre ne peut les exécuter. Autrement dit, la ressource doit être utilisée en *exclusion mutuelle*. La *section critique* est l'ensemble des opérations qui doivent être exécutées en exclusion mutuelle.

▷ **Question 3.2 :**
**Quelle est la section critique pour notre problème du tampon borné ?**

▷ **Question 3.3 :**
**Utilisez un verrou Java pour programmer l'accès exclusif à ces méthodes. Les verrous sont définis dans le paquetage `java.util.concurrent.locks`. L'interface est `Lock` et une réalisation suffisante ici est `ReentrantLock`.**

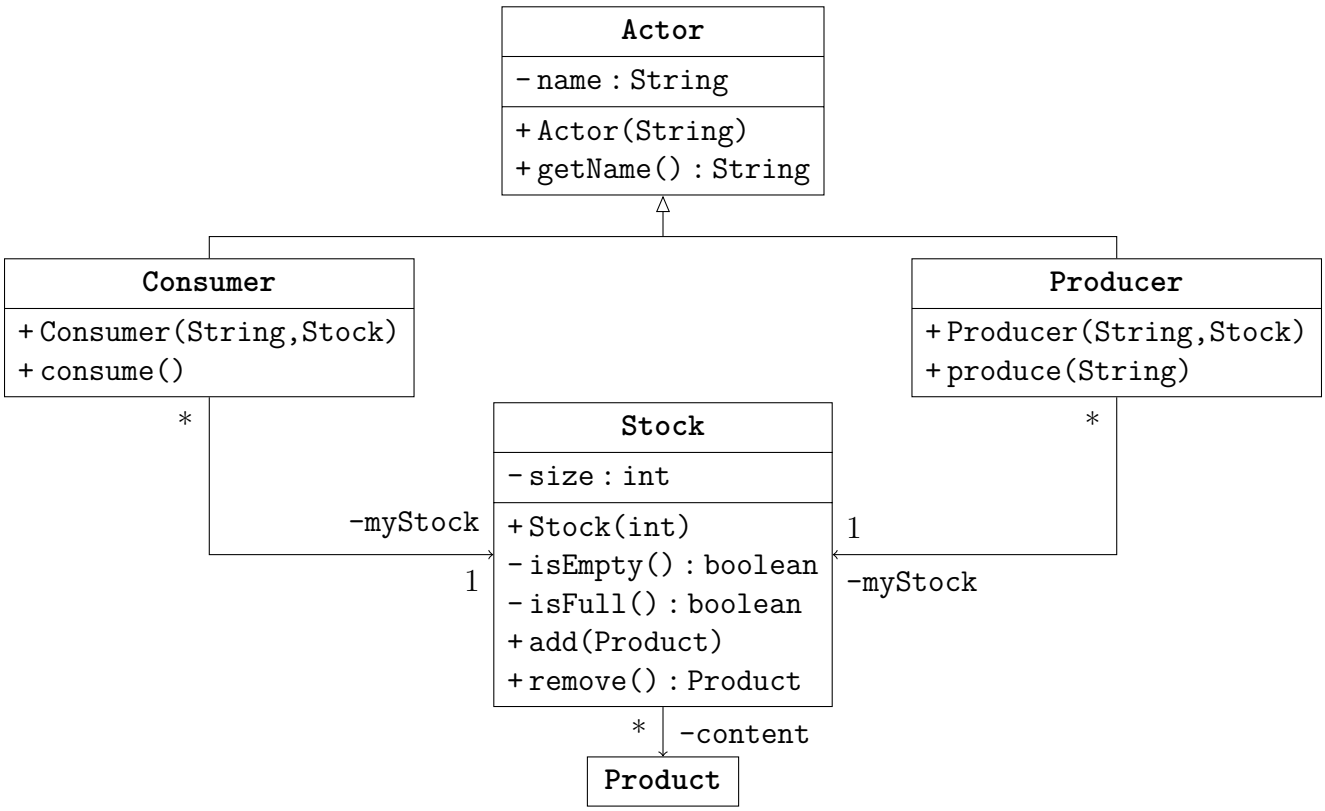


FIGURE 1 – Diagramme de classes d'une solution au problème du tampon borné