4.2 Exemple

Voyons comment mettre en œuvre ces méthodes synchronisées sur l'exemple simple évoqué précédemment (un nombre et son carré). Nous allons donc partager deux informations (*n* et son carré *carre*) entre deux threads. Le premier incrémente *n* et calcule son carré dans *carre* ; le second thread se contente d'afficher le contenu de *carre*.

Ici, les informations sont regroupées dans un objet *nomb* de type *Nombres*. Cette classe dispose de deux méthodes mutuellement exclusives (*synchronized*) :

- *calcul* qui incrémente *n* et calcule la valeur de *carre*,
- *affiche* qui affiche les valeurs de *n* et de *carre*.

Nous créons deux threads de deux classes différentes :

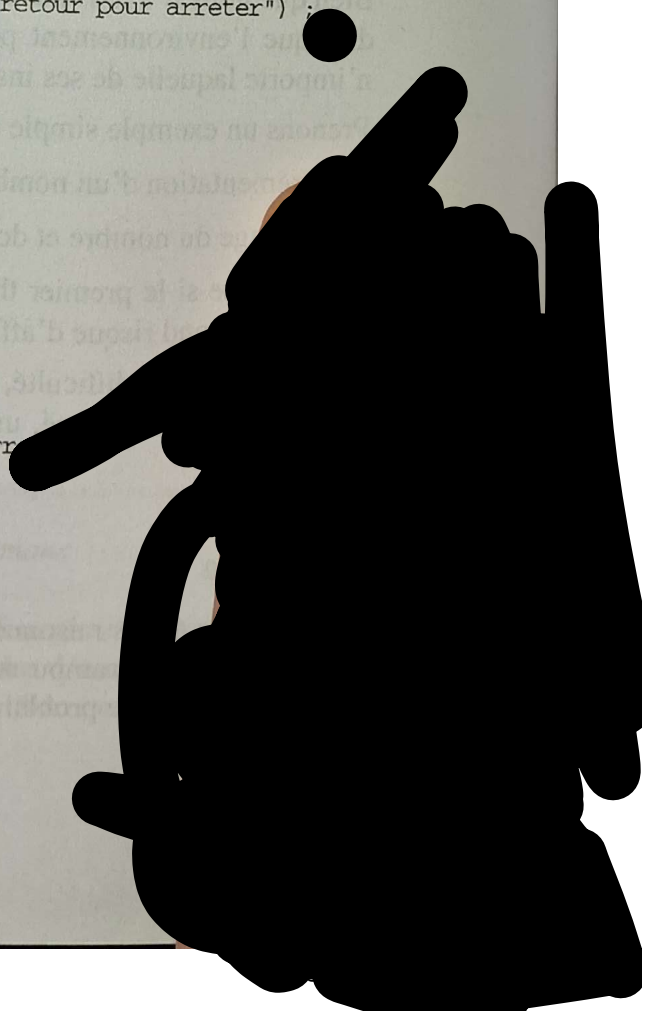
- *calc* de classe *ThrCalc* qui appelle, à son rythme (défini par appel de *sleep*), la méthode *calcul* de *nomb*,
- *aff* de classe *ThrAff* qui appelle, à son rythme (choisi volontairement différent de celui de *calc*), la méthode *affiche* de *nomb*.

Les deux threads sont lancés par *main* et interrompus lorsque l'utilisateur le souhaite (en frappant un texte quelconque).

```
public class Synchr1
{ public static void main (String args[])
  { Nombres nomb = new Nombres() ;
    Thread calc = new ThrCalc (nomb) ;
    Thread aff  = new ThrAff (nomb) ;
    System.out.println ("Suite de carres - tapez retour pour arreter") ;
    calc.start() ; aff.start() ;
    Clavier.lireString() ;
    calc.interrupt() ; aff.interrupt() ;
  }
}

class Nombres
{ public synchronized void calcul()
  { n++ ;
    carre = n*n ;
  }
  public synchronized void affiche ()
  { System.out.println (n + " a pour carre " + carr
  }
  private int n=0, carre ;
}

class ThrCalc extends Thread
{ public ThrCalc (Nombres nomb)
  { this.nomb = nomb ;
  }
```



```
    public void run ()
    { try
      { while (!interrupted())
        { nomb.calcul () ;
          sleep (50) ;
        }
      }
      catch (InterruptedException e) {}
    }
    private Nombres nomb ;
}
class ThrAff extends Thread
{ public ThrAff (Nombres nomb)
  { this.nomb = nomb ;
  }
   public void run ()
  { try
    { while (!interrupted())
      { nomb.affiche() ;
        sleep (75) ;
      }
    }
    catch (InterruptedException e) {}
  }
  private Nombres nomb ;
}
```

```
1 a pour carre 1
2 a pour carre 4
4 a pour carre 16
5 a pour carre 25
7 a pour carre 49
8 a pour carre 64
10 a pour carre 100
11 a pour carre 121
13 a pour carre 169
14 a pour carre 196
16 a pour carre 256
17 a pour carre 289
```

Utilisation de méthodes synchronisées

Remarques

1 Nous ne nous préoccupons pas ici de synchroniser[1] les activités des deux threads ; plus précisément, nous ne cherchons pas à attendre qu'une nouvelle incrémentation ait lieu

avant d'afficher les valeurs, ou qu'un affichage ait eu lieu avant une nouvelle incrémentation.

2 Une méthode synchronisée appartient à un objet quelconque, pas nécessairement à un thread.

3 On peut se demander ce qui se produirait dans l'exemple précédent si les méthodes *calcul* et *affiche* n'avait pas été déclarées *synchronized*. En fait, ici, les méthodes ont une durée d'exécution très brève, de sorte que la probabilité qu'un thread soit interrompu à l'intérieur de l'une d'elles est très faible. Mais nous pouvons accroître artificiellement cette probabilité en ajoutant une attente entre l'incrémentation et le calcul de carré dans *calcul* :

```
        n++ ;
try
{ Thread.sleep (100) ;
}
catch (InterruptedException e) {}
carre = n*n ;
```

L'ensemble du programme ainsi modifié figure parmi les fichiers source disponibles au téléchargement sur www.editions-eyrolles.com sous le nom *Synchr1a.java*.

3 Notion de verrou

À un instant donné, une seule méthode synchronisée peut donc accéder à un objet donné. Pour mettre en place une telle contrainte, on peut considérer que, pour chaque objet doté d'au moins une méthode synchronisée, l'environnement gère un "verrou" (ou une clé) unique permettant l'accès à l'objet. Le verrou est attribué à la méthode synchronisée appelée pour l'objet et il est restitué à la sortie de la méthode. Tant que le verrou n'est pas restitué, aucune autre méthode synchronisée ne peut le recevoir (bien sûr, les méthodes non synchronisées peuvent, quant à elles, accéder à tout moment à l'objet).

Rappelons que ce mécanisme d'exclusion mutuelle est basé sur l'objet lui-même et non sur le thread. Cette distinction pourra s'avérer importante dans une situation telle que la suivante :

```
void synchronized f (...)     // on suppose f appelée sur un objet o
{   .....     // partie I
    g () ;    // appel de g sur le même objet o
    .....     // partie II
}

void g(...)
{  .....                      // g n'est pas synchronisée
}
```

1. Comme nous l'avons déjà évoqué, le mot *synchronized* est quelque peu trompeur.

La méthode *f* (synchronisée), appelée sur un objet *o*, appelle la méthode *g* (non synchronisée) sur le même objet. Le verrou de l'objet *o*, attribué initialement à *f*, est rendu lors de l'appel de *g*. La méthode *g* peut alors se trouver interrompue par un autre thread qui peut modifier l'objet *o*. Ainsi, rien ne garantit que *o* ne sera pas modifié entre l'exécution de *partie I* et celle de *partie II*. En revanche, cette garantie existerait si *g* était elle aussi synchronisée.

Remarque

Ne confondez pas l'exécution de deux méthodes différentes dans un même thread et l'exécution de deux threads différents (qui peuvent éventuellement appeler une même méthode !).

4.4 L'instruction synchronized

Une méthode synchronisée acquiert donc le verrou sur l'objet qui l'a appelée (implicitement) pour toute la durée de son exécution. L'utilisation d'une méthode synchronisée comporte deux contraintes :

- l'objet concerné (celui sur lequel elle acquiert le verrou) est nécessairement celui qui l'a appelée,
- l'objet est verrouillé pour toute la durée de l'exécution de la méthode.

L'instruction *synchronized* permet d'acquérir un verrou sur un objet quelconque (qu'on cite dans l'instruction) pour une durée limitée à l'exécution d'un simple bloc :

```
synchronized (objet)
{ instructions
}
```

L'instruction synchronized

En théorie, on peut faire d'une instruction *synchronized* une méthode (brève) de l'objet concerné. Par exemple, l'instruction précédente pourrait être remplacée par l'appel

```
object.f(...) ;
```

dans lequel *f* serait une méthode de l'objet, réduite au seul bloc *instructions*.

Il y a cependant une exception, à savoir le cas où l'objet concerné est un tableau car on ne peut pas définir de méthodes pour un tableau.

4.5 Interblocage

L'utilisation des verrous sur des objets peut conduire à une situation de blocage connue souvent sous le nom d'"étreinte mortelle" qui peut se définir ainsi :

- le thread *t1* possède le verrou de l'objet *o1* et il attend le verrou de l'objet *o2*,
- le thread *t2* possède le verrou de l'objet *o2* et il attend le verrou de l'objet *o1*.

Comme on peut s'y attendre, Java n'est pas en mesure de détecter ce genre de situation et c'est au programmeur qu'il incombe de gérer cette tâche. À simple titre indicatif, il existe une technique dite d'ordonnancement des ressources qui consiste à numéroter les verrous dans un certain ordre et à imposer aux threads de demander les verrous suivant cet ordre. On évite alors à coup sûr les situations d'interblocage.

4.6 Attente et notification

Comme nous l'avons dit en introduction de ce paragraphe 4, il arrive que l'on ait besoin de coordonner l'exécution de threads, un thread devant attendre qu'un autre ait effectué une certaine tâche pour continuer son exécution.

Là encore, Java offre un mécanisme basé sur l'objet et sur les méthodes synchronisées que nous venons d'étudier :

- une méthode synchronisée peut appeler la méthode *wait* de l'objet dont elle possède le verrou, ce qui a pour effet :
  - de rendre le verrou à l'environnement qui pourra, le cas échéant, l'attribuer à une autre méthode synchronisée,
  - de mettre "en attente" le thread correspondant ; plusieurs threads peuvent être en attente sur un même objet ; tant qu'un thread est en attente, l'environnement ne lui donne pas la main ;
- une méthode synchronisée peut appeler la méthode *notifyAll* d'un objet pour prévenir tous les threads en attente sur cet objet et leur donner la possibilité de s'exécuter (le mécanisme utilisé et le thread effectivement sélectionné pourront dépendre de l'environnement).

Remarque

Il existe également une méthode *notify* qui se contente de prévenir un seul des threads en attente. Son utilisation est fortement déconseillée (le thread choisi dépendant de l'environnement).

Exemple 1

Voici un programme qui gère une "réserve" (de tout ce qui se dénombre). Il comporte :

- un thread qui ajoute une quantité donnée,
- deux threads qui puisent chacun une quantité donnée.

Manifestement, un thread ne peut puiser dans la réserve que si elle contient une quantité suffisante.

La réserve est représentée par un objet *r*, de type *Reserve*. Cette classe dispose de deux méthodes synchronisées *puise* et *ajoute*. Lorsque la méthode *puise* s'aperçoit que la réserve est insuffisante, il appelle *wait* pour mettre le thread correspondant en attente. Parallèlement, la méthode *ajoute* appelle *notifyAll* après chaque ajout.

Les trois threads sont lancés par *main* et interrompus lorsque l'utilisateur le souhaite (en frappant un texte quelconque).

```
public class Synchro3
{ public static void main (String args[])
  { Reserve r = new Reserve () ;
    ThrAjout ta1 = new ThrAjout (r, 100, 15) ;
    ThrAjout ta2 = new ThrAjout (r, 50,  20) ;
    ThrPuise tp =  new ThrPuise  (r, 300, 10) ;
    System.out.println ("Suivi de stock --- faire entree pour arreter ") ;
    ta1.start () ; ta2.start () ; tp.start () ;
    Clavier.lireString() ;
    ta1.interrupt () ; ta2.interrupt () ; tp.interrupt () ;
  }
}
class Reserve extends Thread
{ public synchronized void puise (int v) throws InterruptedException
  { if (v <= stock) { System.out.print ("-- on puise " + v) ;
                      stock -= v ;
                      System.out.println (" et il reste " + stock ) ;
                    }
             else   { System.out.println ("** stock de " + stock
                             + " insuffisant pour puiser " + v ) ;
                      wait() ;
                    }
  }
  public synchronized void ajoute (int v)
  { stock += v ;
    System.out.println ("++ on ajoute " + v
                        + " et il y a maintenant " + stock) ;
    notifyAll() ;
  }
  private int stock = 500 ;    // stock initial = 500
}
class ThrAjout extends Thread
{ public ThrAjout (Reserve r, int vol, int delai)
  { this.vol = vol ; this.r = r ; this.delai = delai ;
  }
  public void run ()
  { try
    { while (!interrupted())
      { r.ajoute (vol) ; sleep (delai) ;
      }
    }
    catch (InterruptedException e) {}
  }
  private int vol ;
  private Reserve r ;
  private int delai ;
}
```

```
class ThrPuise extends Thread
{ public ThrPuise (Reserve r, int vol, int delai)
  { this.vol = vol ; this.r = r ; this.delai = delai ;
  }
  public void run ()
  { try
    { while (!interrupted())
      { r.puise (vol) ;
        sleep (delai) ;
      }
    }
    catch (InterruptedException e) {}
  }
  private int vol ;
  private Reserve r ;
  private int delai ;
}
```

```
-- on puise 300 et il reste 0
** stock de 0 insuffisant pour puiser 300
++ on ajoute 50 et il y a maintenant 50
++ on ajoute 100 et il y a maintenant 150
** stock de 150 insuffisant pour puiser 300
++ on ajoute 50 et il y a maintenant 200
** stock de 200 insuffisant pour puiser 300
++ on ajoute 100 et il y a maintenant 300
-- on puise 300 et il reste 0
** stock de 0 insuffisant pour puiser 300
++ on ajoute 50 et il y a maintenant 50
```

Utilisation de wait *et* notifyAll (1)