Exercice I.1

Question 1. *Que va afficher ce programme lors de son exécution ?*
On aura

```
$ java SyncTest
Hello World $
```

ou

```
$ java SyncTest
World Hello $
```

En effet, il n'y a aucune garantie sur le temps pris par un thread pour démarrer après l'application de la méthode **`start()`**. Le second thread lancé peut être le premier à s'exécuter, puisque les deux threads sont lancés quasiment en même temps.

Les étudiants pourront le voir par eux-mêmes car ce code est disponible dans l'archive `TD_I.zip`.

Question 2. *Comment ajouter un retour à la ligne à la fin de l'exécution du programme ?*

```
public static void main(String [] args) throws InterruptedException {
  Thread t1 = new Thread(new Test("Hello"));
  Thread t2 = new Thread(new Test("World"));
  t1.start();
  t2.start();
  t1.join();
  t2.join();
  System.out.println();
}
```

Il faut attendre que les deux threads aient terminé avant de lancer le retour à la ligne. L'instruction **`join()`** (comme **`wait()`**) peut provoquer une **`InterruptedException`**, que l'on gère habituellement à l'aide d'un bloc **`try ... catch`**. Nous choisissons ici de laisser le **`main()`** propager cette exception.

Exercice I.2

Question 1. *Donner une sortie écran possible de ce programme.*
Il y a deux réponses possibles :

```
[Hello]
[World]
```

ou

```
[World]
[Hello]
```

Dans les deux cas, l'exécution dure deux secondes : le thread qui prend le verrou en premier le conserve pendant sa pause d'une seconde.

Question 2. *Donner une sortie écran possible de ce programme lorsque l'on supprime le mot-clef **`static`** de la définition de l'objet **`unObjet`**.*
Il y a un objet **`unObjet`** par thread : chaque thread prend un verrou intrinsèque différent ; il n'y a pas de conflit. L'exécution globale dure approximativement une seconde. Il y a ici encore deux sorties possibles :

```
[Hello[World]
]
```

ou

```
[World[Hello]
]
```

N.B. Si la question est soulevée, on peut souligner que la sortie

```
[[HelloWorld]
]
```

n'est pas possible car l'instruction **`print()`** est « atomique ».

Une version étendue du code séquentiel étudié dans cet exercice est disponible sur le site de l'UE dans l'archive `TP_A.zip`, car cet exercice correspond au premier exercice du premier TP. Elle ajoute la prise en compte d'un paramètre en entrée, l'affichage de la durée d'exécution et le calcul du taux d'erreur.

Version séquentielle étendue donnée en TP

```
import java.lang.Math;
import java.util.Random;

public class PiSurQuatre {
  public static void main(String[] args) {
    long nbTirages = 1_000_000;     // Précision du calcul, fixée à 1 000 000
    long tiragesDansLeDisque = 0 ;

    if (args.length>0) {            // Si un paramètre est fourni, il détermine nbTirages
      try { nbTirages = 1_000_000 * Integer.parseInt(args[0]); }
      catch(NumberFormatException nfe) {
        System.err.println ("Usage : java PiSurQuatre <nb de tirages (en millions)>");
        System.err.println(nfe.getMessage());
        System.exit(1);
      }
    }

    System.out.println("Nombre de tirages: " + nbTirages/1_000_000 + " million(s).") ;
    final long début = System.nanoTime();

    Random aléa = new Random();
    double x, y;
    for (long i = 0; i < nbTirages; i++) {
      x = aléa.nextDouble() ;
      y = aléa.nextDouble() ;
      if (x * x + y * y <= 1) tiragesDansLeDisque++ ;
    }
    double résultat = (double) tiragesDansLeDisque / nbTirages ;
    System.out.format("Estimation de Pi/4: %.9f %n", résultat) ;
    double erreur = 100 * Math.abs(résultat-Math.PI/4)/(Math.PI/4) ;
    System.out.format("Pourcentage d'erreur: %.9f %% %n", erreur);

    final long fin = System.nanoTime();
    final long durée = (fin - début) / 1_000_000 ;
    System.out.format("Durée du calcul: %.3f s.%n", (double) durée/1000);
  }
}
```

N.B. Selon les mathématiciens, avec un million de tirages, l'erreur observée est en moyenne $\leqslant 0.1$ %. Avec 100 fois plus de tirages, c'est-à-dire 100 millions de tirages, l'erreur doit être dix fois moindre c'est-à-dire $\leqslant 0.01$ %. Exemples d'exécutions observées :

```
$ java PiSurQuatre
Nombre de tirages: 1 million(s).
Estimation de Pi/4: 0,784199000
Pourcentage d'erreur: 0,152682226 %
Durée du calcul: 0,068 s.
$ java PiSurQuatre 100
Nombre de tirages: 100 million(s).
Estimation de Pi/4: 0,785350190
Pourcentage d'erreur: 0,006108163 %
Durée du calcul: 3,950 s.
$ java PiSurQuatre 500
Nombre de tirages: 500 million(s).
Estimation de Pi/4: 0,785362942
Pourcentage d'erreur: 0,004484528 %
Durée du calcul: 19,667 s.
```

Version parallèle également fournie en TP

Plusieurs codes permettent de répondre à l'exercice proposé. Le code ci-dessous est également fourni pour le premier TP. Il crée 10 threads qui se chargent chacun d'un dixième de la tâche globale. Les références des threads créés sont stockées dans un tableau `T`. Chacun des threads compte le nombre de tirages dans le disque observés dans une variable propre : l'attribut **`tiragesDansLeDisque`**, qui est non-statique et volatile.

```
import java.lang.Math;
import java.util.Random;
import java.lang.Thread;

public class PiSurQuatre extends Thread {
  static long nbTirages = 1_000_000;      // Précision du calcul, fixée à 1 000 000
  volatile long tiragesDansLeDisque = 0; // Nb de tirages dans le disque pour chaque thread
  static int nbThreads = 10;              // Nb de threads utilisés
  long part;

  public static void main (String args[]) {
    if (args.length>0) {
      try { nbTirages = 1_000_000 * Integer.parseInt(args[0]); }
      catch(NumberFormatException nfe) {
        System.err.println("Usage : java PiSurQuatre <nb de tirages>");
        System.err.println(nfe.getMessage());
        System.exit(1);
      }
    }

    System.out.println("Nombre de tirages: " + nbTirages/1_000_000 + " million(s).") ;
    final long début = System.nanoTime();

    PiSurQuatre[] T = new PiSurQuatre[nbThreads];
    for(int i=0; i<nbThreads; i++){
      T[i] = new PiSurQuatre(nbTirages / nbThreads);
      T[i].start();
    }
    for(int i=0; i<nbThreads; i++){
      try{ T[i].join(); } catch(InterruptedException e){e.printStackTrace();}
    }
    int somme = 0;
    for(int i=0; i<nbThreads; i++) {
      somme += T[i].tiragesDansLeDisque;
    }
    double résultat = (double) somme / nbTirages ;
    System.out.format("Estimation de Pi/4: %.9f %n", résultat) ;
    double erreur = 100 * Math.abs(résultat-Math.PI/4)/(Math.PI/4) ;
    System.out.format("Pourcentage d'erreur: %.9f %% %n", erreur);

    final long fin = System.nanoTime();
    final long durée = (fin - début) / 1_000_000 ;
    System.out.format("Durée du calcul: %.3f s.%n", (double) durée/1000);
  }

  public PiSurQuatre (long part){
    this.part = part;
  }

  public void run(){
    Random aléa = new Random();
    double x, y;
    for(long i = 0; i <= part; i++){
      x = aléa.nextDouble() ;
      y = aléa.nextDouble() ;
      if (x * x + y * y <= 1) tiragesDansLeDisque++ ;
    }
  }
}
```

Pédagogiquement, cet exercice conduit à utiliser les instructions Java revues dans les deux premiers exercices, en insérant une partie du code parallèle étudié dans le code séquentiel fourni. Autrement dit, l'étudiant doit faire un copier-coller judicieux. Avec un stylo.

On demande ici dix threads pour forcer à utiliser une collection ou un tableau, car on ne va pas définir les threads t1, t2, jusqu'à t10 : c'est peut-être la seule difficulté technique pour les étudiants peu habiles en Java.

Que fait ce code ?

— Il crée et démarre 10 threads de la classe **`PiSurQuatre`** en fournissant au constructeur la valeur de l'attribut **`part`** fixé au dixième du nombre de tirages à réaliser. Notez qu'il sera ainsi facile de modifier le nombre de threads mis à la tâche ultérieurement.

— Les références de ces 10 threads sont stockées dans un tableau **`T`**. Un **`ArrayList`** peut faire aussi bien l'affaire si l'on préfère.

— Il attend ensuite que chaque thread termine, en appliquant la méthode **`join()`** sur chacun d'eux. Il est ici nécessaire de traiter l'exception **`InterruptedException`**.

— Enfin, ce programme calcule et affiche le résultat final.

Il est important de noter que chaque thread comptabilise le nombre de tirages dans le disque à l'aide d'un attribut qui lui est propre ; il faut donc additionner les valeurs de ces 10 compteurs locaux pour obtenir le nombre total de tirages dans le disque afin de calculer et d'afficher le résultat global.

N.B. Contrairement au C, pour les tirages aléatoires, on n'utilisera pas en général une instruction du type

```
alea.setSeed(System.currentTimeMillis());
```

qui permet d'initialiser le générateur aléatoire par une graine déterminée et distincte à chaque exécution, car cet aspect est déjà pris en charge par le constructeur de **`Random`** de Java.

Solution alternative qui sera moins performante

Si l'on veut utiliser un compteur **`tiragesDansLeDisque`** commun aux 10 threads, alors cet attribut doit être déclaré **`static`** et ses incrémentations doivent être synchronisées, c'est-à-dire effectuées en exclusion mutuelle, à l'aide d'un verrou (sinon, on l'a vu en cours, le calcul sera erroné).

```
...
static volatile long tiragesDansLeDisque = 0; // Nb total de tirages dans le disque
...
long somme = PiSurQuatre.tiragesDansLeDisque;
double résultat = (double) somme / nbTirages ;
...
public void run(){
  Random aléa = new Random();
  double x, y;
  for(long i = 0; i <= part; i++){
    x = aléa.nextDouble() ;
    y = aléa.nextDouble() ;
    if (x * x + y * y <= 1) {
      synchronized( PiSurQuatre.class ){tiragesDansLeDisque++ ;}
    }
  }
}
```

Le code obtenu est correct, car les incrémentations de l'attribut **`tiragesDansLeDisque`** sont effectuées de façon « atomique. » Cependant, les étudiants curieux pourront observer en TP que cette alternative est moins performante que le code séquentiel !

Exemple d'exécution observée :

```
$ java PiSurQuatre 500
Nombre de tirages: 500 million(s).
Estimation de Pi/4: 0,785388800
Pourcentage d'erreur: 0,001192185 %
Durée du calcul: 68,174 s.
```

Ce programme nous indique que 7 nains vont être créés et lancés quasiment en même temps. Le constructeur permet d'attribuer un nom aux nains. Chacun d'eux va exécuter la méthode **accéder()** avant d'afficher le message indiquant qu'il a un accès exclusif à la ressource Blanche-Neige. Un premier nain doit pouvoir sortir de la méthode **accéder()** mais les 6 autres doivent y rester *bloqués* tant que le premier n'a pas relâché la ressource.

Nous verrons dans le second cours qu'il existe deux catégories de problèmes de synchronisation : l'*exclusion mutuelle*, qui signifie intuitivement que l'on souhaite être seul au monde (à manipuler certaines données), et l'*attente conditionnelle*, qui correspond aux situations où un thread doit attendre que les conditions deviennent favorables (par exemple dans un système producteur/consommateur). Dans le premier cas, le reflexe consiste à utiliser un verrou. Dans le second, il faut penser à utiliser un système d'attente à base de signaux.

Puisqu'il s'agit de bloquer six nains dans la méthode **accéder()** tant que les conditions ne sont pas favorables, c'est-à-dire tant que Blanche-Neige n'a pas été libérée par le premier nain, nous sommes dans le cas d'un problème de synchronisation à régler à l'aide d'une variable de condition, c'est-à-dire l'instruction **wait()**. Il faudra donc aussi utiliser l'instruction **notify()** ou **notifyAll()** car sinon, en l'état de nos connaissances après le premier cours, aucun thread ne pourrait sortir de l'attente provoquée par **wait()**.

Comment permettre à un premier nain de sortir de la méthode **accéder()** et bloquer les suivants dans cette méthode ? L'idée de cet exercice est que, de proche en proche, en tâtonnant, la solution doit apparaître. Une première ébauche de solution souvent proposée par les étudiants est donnée ci-dessous.

```
  public void accéder() {
    if ( ! libre ) {                    // Le nain s'endort dans l'attente d'un signal
      try { wait(); } catch (InterruptedException e) {e.printStackTrace();}
    }
    libre = false;
    System.out.println(Thread.currentThread().getName() + "␣accède␣à␣la␣ressource.");
  }

  public void relâcher() {
    System.out.println(Thread.currentThread().getName() + "␣relâche␣la␣ressource.");
    libre = true;
    notify();                    // L'un des nains endormis reçoit un signal de réveil
  }
}
```

L'idée est que pour accéder, il faut que la ressource soit libre. Le premier nain qui accède place l'attribut **libre** à faux afin d'obliger les nains suivants à attendre, via l'instruction **wait()**.

Ce code conduira à une erreur à l'exécution, sous la forme d'une exception, car pour exécuter **wait()**, c'est-à-dire **this.wait()**, un thread doit au préalable posséder le verrou intrinsèque de l'objet **this**. De même, pour exécuter **notify()**, c'est-à-dire **this.notify()**, un thread doit au préalable posséder le verrou intrinsèque de l'objet **this**.

Nous écrirons donc plutôt le code ci-dessous :

```
  public void accéder() {
    if ( ! libre ) { // Le nain s'endort dans l'attente d'un signal
      try {
        synchronized(this){this.wait();}
      } catch (InterruptedException e) {e.printStackTrace();}
    }
    libre = false;
    System.out.println(Thread.currentThread().getName() + "␣accède␣à␣la␣ressource.");
  }

  public void relâcher() {
    System.out.println(Thread.currentThread().getName() + "␣relâche␣la␣ressource.");
    libre = true;
    synchronized(this){this.notify();}
  }
}
```

Ce code est cependant erroné : il permet, dans certain cas, aux sept nains d'accéder à la resssource Blanche-Neige en même temps. Il suffit qu'ils exécutent en même temps et au même rythme le code de la méthode **accéder()** : ils constatent alors ensemble que la ressource est libre et décident simultanément de s'en saisir.

Nous sommes ici face à un problème d'exclusion mutuelle : il ne faut pas laisser les nains exécuter le code de la méthode **accéder()** simultanément. La solution naturelle consiste à protéger cette méthode à l'aide d'un

verrou : seul le nain possédant le privilège du verrou pourra exécuter le code de la méthode **accéder()**.

Puisqu'un verrou est déjà utilisé dans le code de la méthode **accéder()**, le plus simple (et le plus prudent) est d'utiliser le même verrou. Nous écrirons ainsi le code ci-dessous :

```
  public void accéder() {
    synchronized(this){
      if ( ! libre ) { // Le nain s'endort dans l'attente d'un signal
        try {
          this.wait(); // Le verrou de this est déjà pris!
      } catch (InterruptedException e) {e.printStackTrace();}
      libre = false;
      System.out.println(Thread.currentThread().getName() + "␣accède␣à␣la␣ressource.");
    }
  }

  public void relâcher() {
    System.out.println(Thread.currentThread().getName() + "␣relâche␣la␣ressource.");
    libre = true;
    synchronized(this){this.notify();}
  }
}
```

Cependant, le code obtenu reste erroné puisqu'il conduira à observer deux nains simultanément en accès à la ressource !

Le scenario catastrophe est le suivant. Lorsque le premier nain ayant accédé à la ressource décide de la relâcher, il relâche également le verrou intrinsèque de **this** après l'envoi du signal. Sur toutes les machines virtuelles testées, ce thread sera prioritaire pour réacquérir le verrou intrinsèque s'il le redemande immédiatement afin d'accéder une seconde fois : il obtiendra ainsi le verrou intrinsèque pour exécuter la méthode **accéder()** et ainsi aura accès une seconde fois d'affilée à la ressource. En quittant la méthode **accéder()** la seconde fois, ce nain libère le verrou intrinsèque. Le nain ayant reçu le signal prendra alors le verrou et finira d'exécuter la méthode **accéder()**, puisqu'il a terminé d'exécuter la ligne

```
  if ( ! libre ) { wait(); }
```

En revanche, en remplaçant cette ligne par

```
  while ( ! libre ) { wait(); }
```

le nain réveillé sera forcé de constater que les conditions ne sont toujours pas favorables, puisque Blanche-Neige n'est plus libre, et qu'il a été réveillé pour rien ; ceci le conduira à retourner en attente en exécutant à nouveau **wait()**.

Ceci conduit à la solution ci-dessous dans laquelle les deux méthodes sont déclarées **synchronized** afin de contraindre les threads qui les exécutent à prendre le verrou de l'objet partagé Blanche-Neige avant d'exécuter leur code.

```
  public synchronized void accéder() {
    while( ! libre ) { // Le nain s'endort dans l'attente d'un signal
      try { wait(); } catch (InterruptedException e) {e.printStackTrace();}
    }
    libre = false;
    System.out.println(Thread.currentThread().getName() + "␣accède␣à␣la␣ressource.");
  }

  public synchronized void relâcher() {
    System.out.println(Thread.currentThread().getName() + "␣relâche␣la␣ressource.");
    libre = true;
    notifyAll();      // Tous les nains endormis reçoivent un signal de réveil
  }
}
```

Ce code sera disponible en ligne dans l'archive `TP_B.zip`. Il est aussi intéressant de souligner que si l'on remplace l'instruction **notify()** par un appel à **notifyAll()**, le code reste correct.

Dans le premier programme à analyser, le Main crée deux threads appelés Cobaye et Observateur, affiche l'état du Cobaye et démarre le Cobaye et l'Observateur. Il attend ensuite que le Cobaye ait terminé puis affiche une seconde fois l'état du Cobaye. Lors du premier affichage, le Cobaye est nécessairement dans l'état NEW puisqu'il n'a pas encore démarré. Lors du second affichage, le Cobaye est nécessairement dans l'état TERMINATED puisqu'il a terminé son code.

Le Cobaye et l'Observateur appellent chacun une fois la méthode **`affiche()`** : il y aura donc au total quatre affichages de l'état du Cobaye. Les deux threads Cobaye et Observateur démarrent quasiment en même temps (et on ne sait pas lequel commencera en premier).

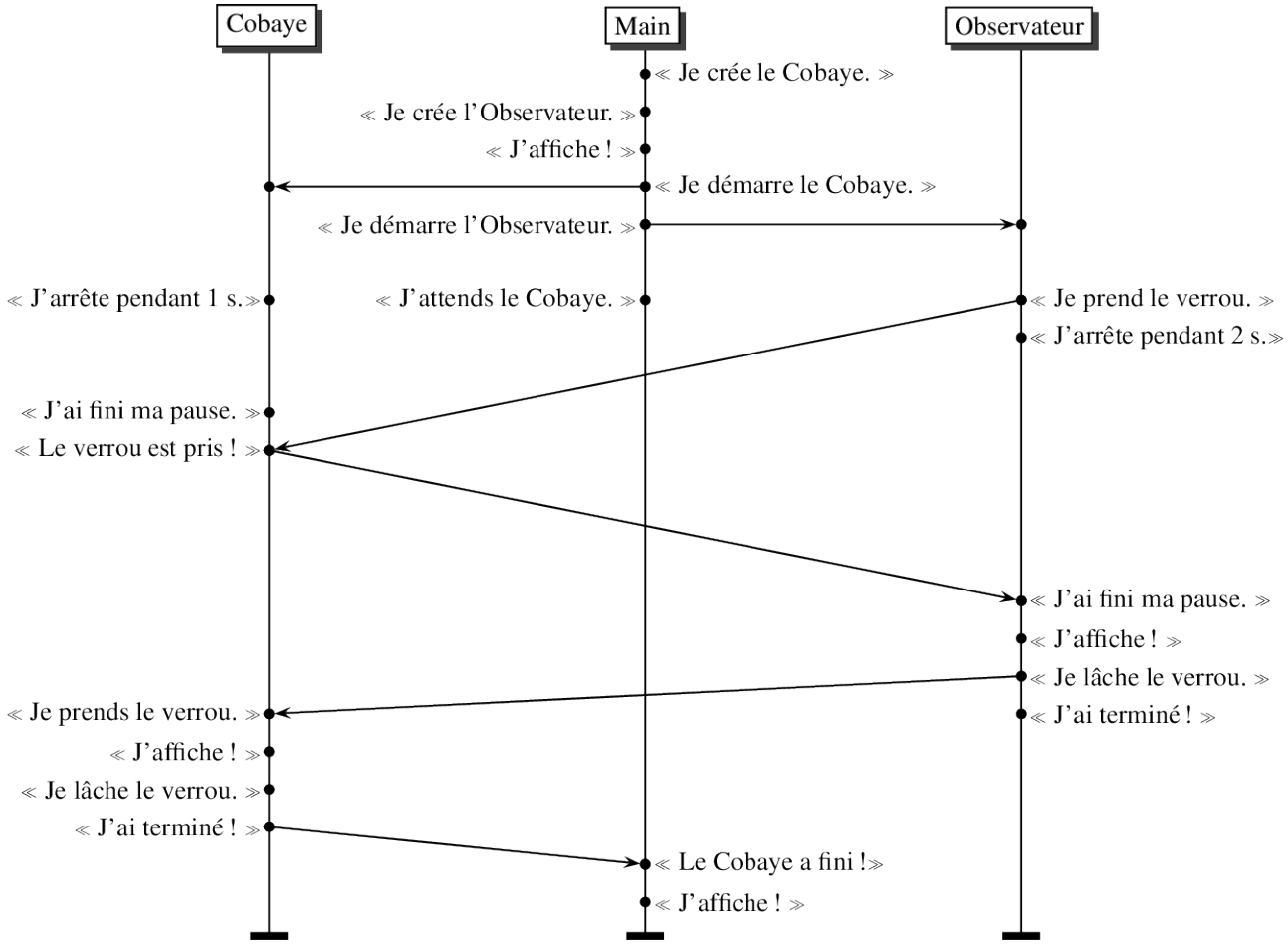
Puisque le Cobaye commence par faire une pause de 1 seconde, l'Observateur a largement le temps de démarrer l'exécution de son code : il prendra donc en premier le verrou intrinsèque de l'objet **`objet`** puis fera une pause de 2 secondes en gardant le verrou.

Au bout d'une seconde, le Cobaye a fini sa pause : il redémarre l'exécution de son code et demande le verrou intrinsèque de l'objet **`objet`**. Celui-ci étant déjà pris par l'Observateur, le Cobaye passe dans l'état BLOCKED. Pendant ce laps de temps, l'Observateur poursuit sa pause.

Au bout d'une seconde, l'Observateur a fini sa pause : il redémarre l'exécution de son code et affiche l'état du Cobaye : c'est BLOCKED. Puis il relâche le verrou intrinsèque de l'objet **`objet`**. Celui-ci est alors acquis par le Cobaye qui quitte l'état BLOCKED pour retourner dans l'état RUNNABLE et poursuivre l'exécution de son code. Il appelle alors la méthode **`Affiche()`** qui indiquera l'état courant du Cobaye : RUNNABLE.

Enfin, le Cobaye ayant terminé l'exécution de son code, il passe dans l'état TERMINATED, ce qui permet au Main de sortir de l'instruction **`join()`** et d'afficher une dernière fois l'état du Cobaye.

Le fonctionnement de ce programme peut donc être schématisé par le diagramme de séquence ci-dessous.



Il ne faudra pas oublier sur la copie d'examen d'indiquer une phrase-réponse, comme à l'école primaire. Nous pouvons conclure que le programme affichera successivement :

— NEW
— BLOCKED
— RUNNABLE
— TERMINATED

Dans le second programme à analyser, le Main est inchangé : il crée les deux threads appelés Cobaye et Observateur, affiche l'état du Cobaye, démarre le Cobaye et l'Observateur. Il attend ensuite que le Cobaye ait terminé puis affiche une seconde fois l'état du Cobaye.

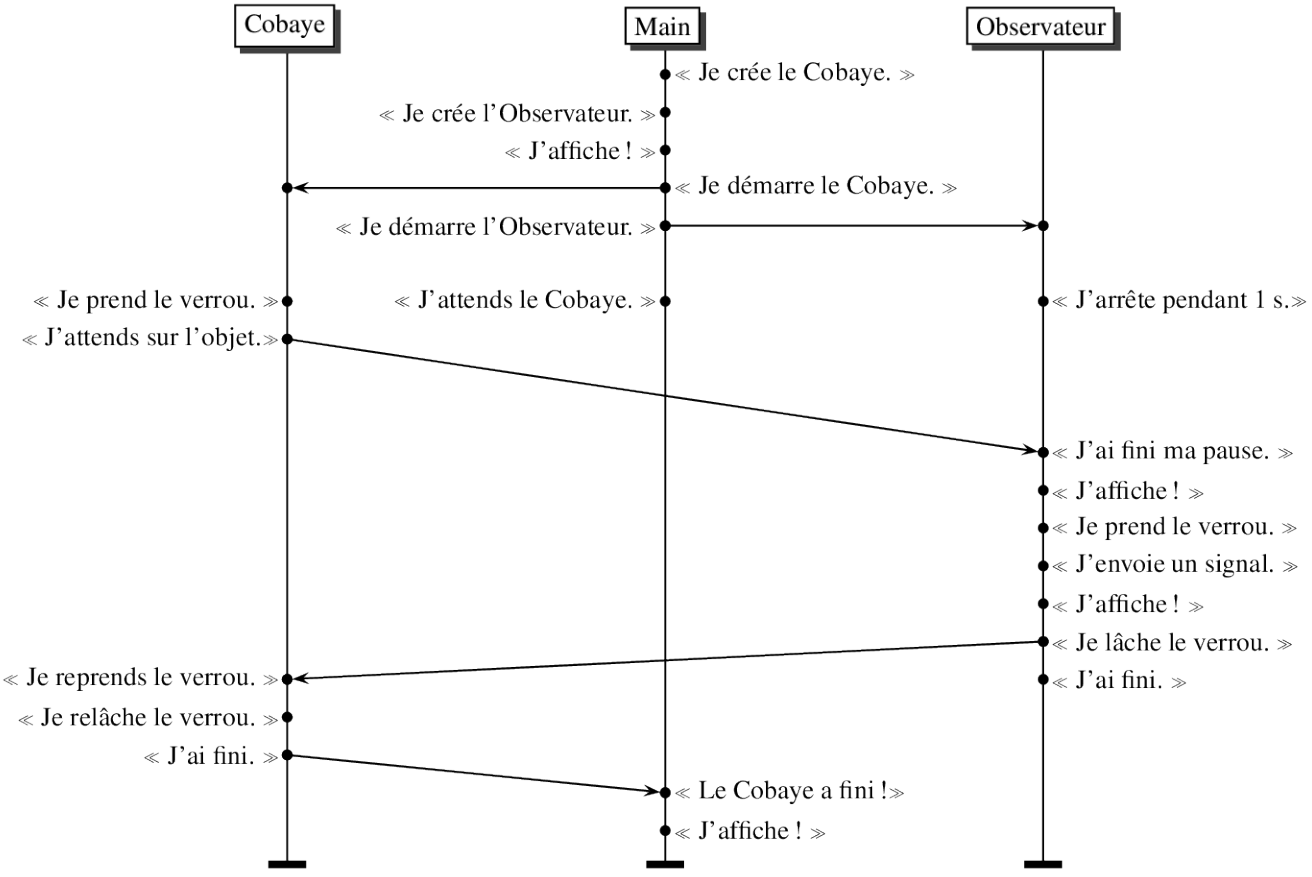
Dans ce second programme, l'Observateur appelle deux fois la méthode **`affiche()`** : il y aura donc au total quatre affichages de l'état du Cobaye, deux par le Main, deux par l'Observateur.

Les deux threads Cobaye et Observateur démarrent quasiment en même temps. Puisque l'Observateur commence par faire une pause de 1 seconde, le Cobaye a largement le temps de prendre le verrou en premier. Il peut alors s'endormir sur l'objet **`objet`**. Il passe alors dans l'état WAITING et relâche le verrou intrinsèque de l'objet **`objet`**. Pendant ce temps, l'Observateur poursuit sa pause.

Au bout d'une seconde, l'Observateur a fini sa pause : il redémarre l'exécution de son code et affiche l'état du Cobaye : c'est WAITING. Puis il prend le verrou intrinsèque de l'objet **`objet`** et envoie un signal sur cet objet. Le Cobaye reçoit ce signal et quitte l'état WAITING pour l'état BLOCKED, car il doit récupérer le verrou avant de poursuivre l'exécution de son code.

Or, l'Observateur conserve le verrou pour afficher l'état du Cobaye une seconde fois : ce sera nécessairement BLOCKED. Puis il relâche le verrou, ce qui permet au Cobaye de prendre le verrou, de retourner dans l'état RUNNABLE et de finir l'exécution de son code. Le Main peut alors à son tour terminer.

Le fonctionnement de ce programme peut donc être schématisé par le diagramme de séquence ci-dessous.



Le second programme proposé affichera successivement :

— NEW
— WAITING
— BLOCKED
— TERMINATED

Ces deux résultats peuvent être vérifiés à l'aide des programmes fournis dans l'archive du TP A.