Exercice I.1 Question d'ordonnancement

```
class Test implements Runnable {
    String msg;

    public Test(String s) {
        msg = s;
    }

    public void run() {
        try { Thread.sleep(1000); } catch (InterruptedException e) {e.printStackTrace();};
        System.out.print(msg + "␣");
    }

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

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

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

Exercice I.2 Fonctionnement d'un verrou

```
class SyncTest extends Thread {
  String msg;
  private static Object unObjet = new Object(); // Utilisé pour son verrou intrinsèque

  public SyncTest(String s) {
    msg = s;
  }

  public void run() {
    synchronized (unObjet) {
      System.out.print("[" + msg);
      try { sleep(1000); } catch (InterruptedException e) {e.printStackTrace();}
      System.out.println("]");
    }
  }

  public static void main(String [] args) {
    new SyncTest("Hello").start();
    new SyncTest("World").start();
  }
}
```

Question 1. Donner une sortie écran possible de ce programme.

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`**.

Exercice I.3 Evaluation de $\pi/4$ par la méthode de Monte-Carlo La méthode dite « de Monte-Carlo » consiste à calculer une valeur numérique en utilisant des procédés aléatoires, c'est-à-dire des techniques probabilistes. Considérons par exemple un point M de coordonnées (x, y) avec $0 < x < 1$ et $0 < y < 1$ tirées aléatoirement. Le point M appartient au disque de centre (0,0) de rayon 1 si, et seulement si, $x^2 + y^2 \leqslant 1$. La probabilité que le point M appartienne au disque est donc de $\pi/4$. Si l'on effectue un grand nombre de tirages de points, le rapport du nombre de points dans le disque au nombre de tirages total fournit une approximation du nombre $\pi/4$. Cette idée conduit au code du programme Java **`PiSurQuatre`** de la figure 1 qui évalue et affiche une estimation de la valeur de $\pi/4$.

Pour accélérer ce calcul sur une machine multi-coeur, on souhaite le paralléliser sur 10 threads. Écrire un programme Java qui crée les 10 threads et leur affecte une part équitable de la tâche globale à effectuer ; attend que tous les threads aient terminé leur tâche ; puis affiche la valeur approchée de $\pi/4$. Ce programme ne doit définir *qu'une seule classe de threads (ou de runnables).*

2014

2017

```
public class PiSurQuatre {
  public static void main(String[] args) {
    int nbTirages = 1_000_000;
    int tiragesDansLeDisque = 0;
    double x, y, résultat;
    Random alea = new Random();
    for (int i = 0; i < nbTirages; i++) {
      x = alea.nextDouble();
      y = alea.nextDouble();
      if (x * x + y * y <= 1) tiragesDansLeDisque++;
    }
    résultat = (double) tiragesDansLeDisque/nbTirages;
    System.out.println("Estimation␣de␣Pi/4:␣" + résultat);
  }
}
```

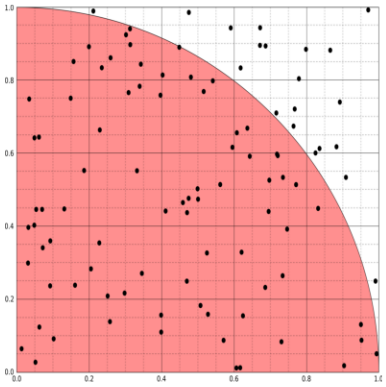

FIGURE 1 – Calcul approché de $\pi/4$ par la méthode de Monte-Carlo

```
public class SeptNains {
  public static void main(String[] args) {
    int nbNains = 7;
    String nom [] = {"Simplet", "Dormeur", "Atchoum", "Joyeux", "Grincheux", "Prof",
      "Timide"};
    Nain nain [] = new Nain [nbNains];
    for(int i = 0; i < nbNains; i++) nain[i] = new Nain(nom[i]);
    for(int i = 0; i < nbNains; i++) nain[i].start();
  }
}

class BlancheNeige {
  private volatile boolean libre = true;   // Initialement, Blanche-Neige est libre.
  public void requérir() {
    System.out.println(Thread.currentThread().getName() + "␣veut␣un␣accès␣exclusif.");
  }
  public void accéder() {    // COMPLÉTEZ CETTE MÉTHODE
    ...
    System.out.println(Thread.currentThread().getName() + "␣accède␣à␣la␣ressource.");
    ...
  }

  public void relâcher() { // COMPLÉTEZ CETTE MÉTHODE
    ...
    System.out.println(Thread.currentThread().getName() + "␣relâche␣la␣ressource.");
    ...
}

class Nain extends Thread {
  private static final BlancheNeige bn = new BlancheNeige();
  public Nain(String nom) {
    this.setName(nom);
  }
  public void run() {
    while(true) {
      bn.requérir();
      bn.accéder();
      System.out.println(getName() + "␣a␣un␣accès␣exclusif␣à␣Blanche-Neige.");
      try { sleep(1000); } catch (InterruptedException e) {e.printStackTrace();}
      bn.relâcher();
    }
  }
}
```

FIGURE 2 – Programme des sept nains à compléter (Exercice I.4)

Exercice I.4 Synchronisation de nains (avec attente passive) Il s'agit dans cet exercice de compléter le programme `SeptNains.java` la figure 2. Le **`main`** lance 7 threads (les 7 nains) qui entrent en concurrence pour l'accès *exclusif* à une ressource partagée (Blanche-Neige, notée **`bn`**). Le comportement d'un nain est le suivant :

① Au démarrage, le nain demande à avoir accès à la ressource, en affichant une requête sur la sortie standard ;

② Il affiche également un message de bienvenue lorsqu'il accède à la ressource. Puis il patiente une seconde.

③ Enfin, il relâche la ressource et affiche un message d'adieu. Il tente alors immédiatement d'accéder à nouveau à la ressource, sans fin.

Complétez la classe **`BlancheNeige`** en proposant un codage pour les deux méthodes **`accéder()`** et **`relâcher()`** de sorte que jamais deux nains aient simultanément accès à la ressource. Chacune de ces méthodes se charge des affichages à l'écran du nom du nain en train de l'exécuter (sur le modèle de la méthode **`requérir()`**). En outre, *l'attente active est proscrite* : dès lors qu'un thread doit patienter jusqu'à ce que les conditions soient plus favorables, il faudra qu'il exécute l'instruction **`wait()`**.

Exercice I.5 L'état du cobaye Devinez l'affichage de chaque appel à la méthode **`Affiche()`** dans le programme ci-dessous.

2017

```
class IntrinsicLock {
  static Object objet = new Object();
  static Thread Cobaye, Observateur;
  public static void Affiche(){
    System.out.print("Etat du thread Cobaye: " + Cobaye.getState());
  }
  public static void main(String[] args) throws InterruptedException {
    Cobaye = new Thread( new Runnable() { public void run() {
        try{ Thread.sleep(1000); } catch(InterruptedException e){e.printStackTrace();}
        synchronized(objet){ // Le Cobaye a pris le verrou
          Affiche();
        }
      }
    }
    );
    Observateur = new Thread( new Runnable(){ public void run(){
        synchronized(objet){
          try{ Thread.sleep(2000); } catch(InterruptedException e){e.printStackTrace();}
          Affiche();
        }
      }
    }
    );
    Affiche();
    Cobaye.start();
    Observateur.start();
    Cobaye.join();
    Affiche();
    Observateur.join();
  }
}
```

Même question avec la variante ci-dessous :

2016

```
Cobaye = new Thread(new Runnable() { public void run() {
    synchronized(objet) {
      try { objet.wait() ; } catch(InterruptedException e){e.printStackTrace();}
    }
  }
});
Observateur = new Thread(new Runnable() { public void run() {
    try { Thread.sleep(1000); } catch(InterruptedException e){e.printStackTrace();}
    Affiche( );
    synchronized(objet){
      objet.notify() ;
      Affiche() ;
    }
  }
});
```