Esercizio 1

Il programma è composto da 2 threads: il primo scrive a schermo il valore attuale della variabile "sum" (che viene incrementata dall'altro thread) e conteggia quante volte riesce a scrivere a schermo. Invece, il secondo thread incrementa la variabile "sum" e avvisa il primo thread che ha terminato l'esecuzione, modificando a true il valore della variabile "finished". Per entrabi i thread le variabili vengono incrementate localmente (stack confinement) ed in seguito condivise tramite variabile volatile, in modo da evitare race-conditions di tipo read-modify-write. Inoltre, per sincronizzare l'esecuzione dei thread viene sfruttato un CountDownLatch. Senza modifiche al programma, l'intervento dello scheduler del sistema operativo è raro e può capitare che un thread esegua per diverso tempo senza bloccarsi. Questo tipo di comportamento può limitare le possibilità che hanno gli altri thread di eseguire. La chiamata al metodo Thread.yield() permette di modificare questo comportamento, agevolando l'esecuzione dello scheduler. Di conseguenza, gli altri thread possono eseguire più frequentemente. In alternativa, si possono modificare le priorità dei threads, favorendo l'esecuzione dei thread coinvolti nella visualizzazione. Come si può vedere dalla tabella sottostante, le due soluzioni proposte permettono d'aumentare il numero di messaggi scritti a schermo dal programma.

	Original	Thread.yield	Thread priority
Count	40	387	60

```java
// Thread.yield()
public class S9Esercizio1 extends Thread {
  static volatile boolean finished = false;
  static volatile int sum = 0;
  static volatile int cnt = 0;
  static final CountDownLatch cdl = new
CountDownLatch(2);

  public static void main(final String[] args)
{
    final Thread thread1 = new Thread(() -> {
    cdl.countDown();
    try {
      cdl.await();
    } catch (final InterruptedException e) {
      e.printStackTrace();
    }
    int count = 0;
    while (!S9Esercizio1.finished) {
      S9Esercizio1.cnt = ++count;
System.out.println("sum " + S9Esercizio1.sum);
    }
  });

    final Thread thread2 = new Thread(() -> {
      cdl.countDown();
      for (int i = 1; i <= 50000; i++) {
        S7Esercizio1.sum = i;
        Thread.yield(); // yield version
      }
      S9Esercizio1.finished = true;
System.out.println("Cnt " + S9Esercizio1.cnt);
    });



    thread1.start();
    thread2.start();

    try {
      thread1.join();
      thread2.join();
    } catch (final InterruptedException e) {
      e.printStackTrace();
    }}}
```

```java
// Thread priority
public class S9Esercizio1 extends Thread {
  static volatile boolean finished = false;
  static volatile int sum = 0;
  static volatile int cnt = 0;
  static final CountDownLatch cdl = new
CountDownLatch(2);

  public static void main(final String[] args)
{
    final Thread thread1 = new Thread(() -> {
      cdl.countDown();
    try {
      cdl.await();
    } catch (final InterruptedException e) {
      e.printStackTrace();
    }
      int count = 0;
      while (!S9Esercizio1.finished) {
        S9Esercizio1.cnt = ++count;
System.out.println("sum " + S9Esercizio1.sum);
      }
    });

    final Thread thread2 = new Thread(() -> {
      cdl.countDown();
      for (int i = 1; i <= 50000; i++) {
        S7Esercizio1.sum = i;

      }
      S9Esercizio1.finished = true;
System.out.println("Cnt: "+ S9Esercizio1.cnt);
    });

    // Priority version
    thread2.setPriority(Thread.MAX_PRIORITY);

    thread1.start();
    thread2.start();

    try {
      thread1.join();
      thread2.join();
    } catch (final InterruptedException e) {
      e.printStackTrace();
    }}
```

Esercizio 2

Il programma soffre di un problema di deadlock, causato dal fatto che l'ordine d'acquisizione dei lock è casuale e non previene la situazione in cui thread diversi richiedano gli stessi lock ma in ordine inverso. Per risolvere il problema è sufficiente stabilire un ordine di acquisizione dei vari locks comune a tutti i threads, ad esempio ordinando, in maniera crescente, gli oggetti Depot secondo il loro ID.

```
class AssemblingWorker implements Runnable {
  private final int id;

  public AssemblingWorker(final int id) {
    this.id = id;
  }

  @Override
  public void run() {
    final Random random = new Random();
    int failureCounter = 0;
    while (true) {
      // Choose randomly 3 different suppliers
      final List<Depot> depots = new ArrayList<>();
      while (depots.size() != 3) {
        final Depot randomDepot = S8Factory.suppliers[random.nextInt(S8Factory.suppliers.length)];
        if (!depots.contains(randomDepot))
          depots.add(randomDepot);
      }

      // Sort suppliers to avoid deadlock due to lock order!
      Collections.sort(depots, (d1, d2) -> d1.getId() - d2.getId());

      final Depot supplier1 = depots.get(0);
      final Depot supplier2 = depots.get(1);
      final Depot supplier3 = depots.get(2);

      log("assembling from : " + supplier1 + ", " + supplier2 + ", " + supplier3);
      synchronized (supplier1) {
        synchronized (supplier2) {
          synchronized (supplier3) {
            if (supplier1.isEmpty() || supplier2.isEmpty() || supplier3.isEmpty()) {
              log("not all suppliers have stock available!");
              failureCounter++;
            } else {
              final String element1 = supplier1.getElement();
              final String element2 = supplier2.getElement();
              final String element3 = supplier3.getElement();
              log("assembled product from parts:  " + element1 + ", " + element2 + ", " + element3);
            }
          }
        }
      }

      if (failureCounter > 1000) {
        log("Finishing after " + failureCounter + " failures");
        break;
      }
    }
  }

  private final void log(final String msg) {
    System.out.println("AssemblingWorker" + id + ": " + msg);
  }
}
```

Esercizio 3

La soluzione proposta è quella presentata durante la lezione, che prevede l'utilizzo di un lock unico per ovviare al problema di deadlock e dell'introduzione dello stato *HUNGRY* per evitare l'insorgere di starvation.

```
class Fork {
  public static final char FORK = '|';
  public static final char NO_FORK = ' ';
  int id;
  boolean taken = false;

  public Fork(final int id) {
    this.id = id;
  }

  public boolean take() {
    if (!taken) {
      taken = true;
      return true;
    }
    return false;
  }

  public void release() {
    taken = false;
  }
}

class Philosopher extends Thread {
  public static final char PHIL_THINKING = '-';
  public static final char PHIL_HUNGRY = 'H';
  public static final char PHIL_EATING = 'o';
  private final int id;

  public Philosopher(final int id) {
    this.id = id;
  }

  @Override
  public void run() {
    final Random random = new Random();
    final int tableOffset = 4 * id;
    final Fork leftFork = S9Philosophers.listOfLocks[id];
    final Fork rightFork = S9Philosophers.listOfLocks[(id + 1) % S9Philosophers.NUM_PHILOSOPHERS];
    final int leftPhilo = (id == 0) ? S9Philosophers.NUM_PHILOSOPHERS - 1 : (id - 1);
    final int rightPhilo = ((id + 1) % S9Philosophers.NUM_PHILOSOPHERS);

    final int table__farL = tableOffset + 0;
    final int table__left = tableOffset + 1;
    final int table_philo = tableOffset + 2;
    final int table_right = tableOffset + 3;
    final int table__farR = (tableOffset + 4) % (4 * S9Philosophers.NUM_PHILOSOPHERS);

    while (!isInterrupted()) {
      try {
        Thread.sleep(S9Philosophers.UNIT_OF_TIME * (random.nextInt(6)));
      } catch (final InterruptedException e) {
        break;
      }

      boolean done = false;
      synchronized (S9Philosophers.class) {
        // Set HUNGRY state
        S9Philosophers.dinerTable[table_philo] = PHIL_HUNGRY;

        // Try to take left fork
        if (leftFork.take()) {
          S9Philosophers.dinerTable[table__farL] = Fork.NO_FORK;
          S9Philosophers.dinerTable[table__left] = Fork.FORK;
          // Try to take right fork
          if (rightFork.take()) {
            done = true;
            S9Philosophers.dinerTable[table_philo] = PHIL_EATING;
```

```java
          S9Philosophers.dinerTable[table_right] = Fork.FORK;
          S9Philosophers.dinerTable[table__farR] = Fork.NO_FORK;
        } else {
          // Could not take right fork: release left fork
          leftFork.release();
          S9Philosophers.dinerTable[table__farL] = Fork.FORK;
          S9Philosophers.dinerTable[table__left] = Fork.NO_FORK;
        }
      }

      // Failed to get left or right fork. Wait until both are available
      if (!done) {
        try {
          while (S9Philosophers.dinerTable[table_philo] != PHIL_EATING) {
            S9Philosophers.class.wait();
          }

          // Update representation with both forks
          S9Philosophers.dinerTable[table__farL] = Fork.NO_FORK;
          S9Philosophers.dinerTable[table__left] = Fork.FORK;
          S9Philosophers.dinerTable[table_philo] = PHIL_EATING;
          S9Philosophers.dinerTable[table_right] = Fork.FORK;
          S9Philosophers.dinerTable[table__farR] = Fork.NO_FORK;
        } catch (final InterruptedException e) {
          break;
        }
      }
    }

    // Eat
    try {
      sleep(S9Philosophers.UNIT_OF_TIME * 1);
    } catch (final InterruptedException e) {
      break;
    }

    // Put forks on the table and go back thinking
    synchronized (S9Philosophers.class) {
      leftFork.release();
      S9Philosophers.dinerTable[table__farL] = Fork.FORK;
      S9Philosophers.dinerTable[table__left] = Fork.NO_FORK;

      rightFork.release();
      S9Philosophers.dinerTable[table_right] = Fork.NO_FORK;
      S9Philosophers.dinerTable[table__farR] = Fork.FORK;

      S9Philosophers.dinerTable[table_philo] = PHIL_THINKING;

      checkAndResume(leftPhilo);
      checkAndResume(rightPhilo);
    }
  }
}

private static void checkAndResume(final int neighborPhilosopher) {
  final int tableOffset = neighborPhilosopher * 4;
  final int leftPhilo;
  if ((tableOffset - 2) < 0)
    leftPhilo = (4 * S9Philosophers.NUM_PHILOSOPHERS - 2);
  else
    leftPhilo = (tableOffset - 2);
  final int rightPhilo = (tableOffset + 6) % (4 * S9Philosophers.NUM_PHILOSOPHERS);
  final int table_philo = tableOffset + 2;

  if (S9Philosophers.dinerTable[table_philo] == PHIL_HUNGRY
      && S9Philosophers.dinerTable[leftPhilo] != PHIL_EATING
      && S9Philosophers.dinerTable[rightPhilo] != PHIL_EATING) {
    S9Philosophers.dinerTable[table_philo] = PHIL_EATING;
    final Fork leftFork = S9Philosophers.listOfLocks[neighborPhilosopher];
    final Fork rightFork = S9Philosophers.listOfLocks[(neighborPhilosopher + 1)
                                                      % S9Philosophers.NUM_PHILOSOPHERS];
    leftFork.take();
    rightFork.take();
    S9Philosophers.class.notifyAll();
  }
}
```

```java
}

public class S9Philosophers {
  public static final int NUM_PHILOSOPHERS = 5;
  public static final int UNIT_OF_TIME = 50;
  public static final Fork[] listOfLocks = new Fork[NUM_PHILOSOPHERS];
  public static char[] dinerTable = null;

  static {
    for (int i = 0; i < NUM_PHILOSOPHERS; i++)
      listOfLocks[i] = new Fork(i);
  }

  public static void main(final String[] a) {
    final char[] lockedDiner = new char[4 * NUM_PHILOSOPHERS];
    for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
      lockedDiner[4 * i + 0] = Fork.NO_FORK;
      lockedDiner[4 * i + 1] = Fork.FORK;
      lockedDiner[4 * i + 2] = Philosopher.PHIL_HUNGRY;
      lockedDiner[4 * i + 3] = Fork.NO_FORK;
    }
    final String lockedString = new String(lockedDiner);

    // safe publication of the initial representation
    synchronized (S9Philosophers.class) {
      dinerTable = new char[4 * NUM_PHILOSOPHERS];
      for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
        dinerTable[4 * i + 0] = Fork.FORK;
        dinerTable[4 * i + 1] = Fork.NO_FORK;
        dinerTable[4 * i + 2] = Philosopher.PHIL_THINKING;
        dinerTable[4 * i + 3] = Fork.NO_FORK;
      }
    }

    for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
      final Thread t = new Philosopher(i);
      // uses this solution to allow terminating the application even if
      // there is a deadlock
      t.setDaemon(true);
      t.start();
    }

    System.out.println("The diner table:");
    long step = 0;
    while (true) {
      step++;

      String curTableString = null;
      synchronized (S9Philosophers.class) {
        curTableString = new String(dinerTable);
      }
      System.out.println(curTableString + "   " + step);

      if (lockedString.equals(curTableString))
        break;
      try {
        Thread.sleep(UNIT_OF_TIME);
      } catch (final InterruptedException e) {
        System.out.println("Interrupted.");
      }
    }
    System.out.println("The diner is locked.");
  }
}
```

Esercizio 4

La seguente soluzione al problema del barbiere è stata sviluppata usando le blocking queues.

```java
import java.util.ArrayList;
import java.util.List;
import java.util.Random;
import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.atomic.AtomicInteger;

class QueueCustomer implements Runnable {
  public static final QueueCustomer LAST = new QueueCustomer(-1);
  int iD;

  public QueueCustomer(final int i) {
    iD = i;
  }

  @Override
  public void run() {
    final Random random = new Random();
    // Enter shop
    if (BlockingQueueSleepingBarber.cuttingChairs.offer(this)) {
      BlockingQueueSleepingBarber.customerCuttingChair.incrementAndGet();
      log("sitting on cutting seat.");
    } else {
      log("no free cutting seats. Going to the waiting room.");

      try {
        Thread.sleep(random
            .nextInt(BlockingQueueSleepingBarber.CUSTOMER_TIME_TO_WAIT_ROOM_MAX
                - BlockingQueueSleepingBarber.CUSTOMER_TIME_TO_WAIT_ROOM_MIN)
            + BlockingQueueSleepingBarber.CUSTOMER_TIME_TO_WAIT_ROOM_MIN);
      } catch (final InterruptedException e) {
      }

      // Try to enter the waitingRoom
      if (BlockingQueueSleepingBarber.waitingChairs.offer(this)) {
        BlockingQueueSleepingBarber.customerWaitingChair.incrementAndGet();
        waitForHaircut();
        log("leaving the waitingRoom");
      } else {
        BlockingQueueSleepingBarber.customerLeft.incrementAndGet();
        log("leaving the barbershop because there are no free waiting seats.");
      }
    }
  }

  // take a seat
  public void getHaircut() {
    log("is getting it's haircut");
  }

  public synchronized void awake() {
    notify();
  }

  private synchronized void waitForHaircut() {
    while (BlockingQueueSleepingBarber.waitingChairs.contains(this)) {
      log("waiting in the waitingRoom");
      try {
        wait();
      } catch (final InterruptedException e) {
        e.printStackTrace();
      }
    }
  }

  private void log(final String msg) {
    System.out.println(this + ": " + msg);
  }

  @Override
  public String toString() {
```

```java
    return "Customer" + iD;
  }
}

class QueueBarber implements Runnable {
  private final Random random = new Random();

  @Override
  public void run() {
    while (true) {
      QueueCustomer customer;
      try {
        // Take: wait until customer becomes available
        customer = BlockingQueueSleepingBarber.cuttingChairs.take();
        if (customer != null && customer.equals(QueueCustomer.LAST)) {
          System.out.println("QueueBarber is leaving");
          return;
        }

        if (customer == null) {
          goToWaitingRoom();
          customer = BlockingQueueSleepingBarber.waitingChairs.poll();
        }

        if (customer != null) {
          cutHair(customer);

          customer = BlockingQueueSleepingBarber.waitingChairs.poll();
          if (customer != null) {
            customer.awake();
            cutHair(customer);
          } else {
            System.out.println("QueueBarber is going to sleep");
            BlockingQueueSleepingBarber.barberSleeps.incrementAndGet();
          }
        }
      } catch (final InterruptedException e) {
        e.printStackTrace();
      }
    }
  }

  private void goToWaitingRoom() throws InterruptedException {
    Thread.sleep(random
        .nextInt(BlockingQueueSleepingBarber.BARBER_TIME_TO_WAIT_ROOM_MAX
            - BlockingQueueSleepingBarber.BARBER_TIME_TO_WAIT_ROOM_MIN)
        + BlockingQueueSleepingBarber.BARBER_TIME_TO_WAIT_ROOM_MIN);
  }

  // simulate cutting hair
  public void cutHair(final QueueCustomer customer) {
    BlockingQueueSleepingBarber.barberNumCuts.incrementAndGet();
    customer.getHaircut();
    System.out.println("The barber is cutting hair for " + customer);

    try {
      Thread.sleep(random
          .nextInt(BlockingQueueSleepingBarber.BARBER_TIME_TO_CUT_MAX
              - BlockingQueueSleepingBarber.BARBER_TIME_TO_CUT_MIN)
          + BlockingQueueSleepingBarber.BARBER_TIME_TO_CUT_MIN);
    } catch (final InterruptedException e) {
        e.printStackTrace();
    }
  }
}

public class BlockingQueueSleepingBarber {
  public static final int CUTTING_CHAIRS = 1;
  public static final int WAITING_CHAIRS = 1;
  public static final int NUM_CUSTOMERS = 100;
  public static final int BARBER_TIME_TO_CUT_MIN = 500;
  public static final int BARBER_TIME_TO_CUT_MAX = 1000;
  public static final int BARBER_TIME_TO_WAIT_ROOM_MIN = 50;
  public static final int BARBER_TIME_TO_WAIT_ROOM_MAX = 100;
  public static final int CUSTOMER_TIME_TO_ENTER_MIN = 450;
  public static final int CUSTOMER_TIME_TO_ENTER_MAX = 700;
```

```
  public static final int CUSTOMER_TIME_TO_WAIT_ROOM_MIN = 80;
  public static final int CUSTOMER_TIME_TO_WAIT_ROOM_MAX = 160;

  public static BlockingQueue<QueueCustomer> cuttingChairs = new
ArrayBlockingQueue<QueueCustomer>(CUTTING_CHAIRS);
  public static BlockingQueue<QueueCustomer> waitingChairs = new
ArrayBlockingQueue<QueueCustomer>(WAITING_CHAIRS);

  // Verification counters
  public static AtomicInteger barberNumCuts = new AtomicInteger(0);
  public static AtomicInteger barberSleeps = new AtomicInteger(0);
  public static AtomicInteger customerCuttingChair = new AtomicInteger(0);
  public static AtomicInteger customerWaitingChair = new AtomicInteger(0);
  public static AtomicInteger customerLeft = new AtomicInteger(0);

  public static void main(final String args[]) {
    final Random random = new Random();
    final List<Thread> allCustomers = new ArrayList<>();
    final Thread barber = new Thread(new QueueBarber(), "Barber");
    barber.start();

    // create new customers
    for (int i = 1; i <= NUM_CUSTOMERS; i++) {
      final Thread newCustomer = new Thread(new QueueCustomer(i), "Customer" + i);
      allCustomers.add(newCustomer);
      newCustomer.start();
      try {
        Thread.sleep(random.nextInt(CUSTOMER_TIME_TO_ENTER_MAX - CUSTOMER_TIME_TO_ENTER_MIN)
            + CUSTOMER_TIME_TO_ENTER_MIN);
      } catch (final InterruptedException ex) {
         ex.printStackTrace();
      }
    }
    // Add Special Customer to queue to terminate the barber's thread
    try {
      cuttingChairs.put(QueueCustomer.LAST);
    } catch (final InterruptedException e1) {
      e1.printStackTrace();
    }

    // Wait for the barber to quit
    try {
      barber.join();
    } catch (final InterruptedException e) {
        e.printStackTrace();
    }

    // Wait for all customers to quit
    for (final Thread customerThread : allCustomers) {
      try {
        customerThread.join();
      } catch (final InterruptedException e) {
        e.printStackTrace();
      }
    }

 // Verify
 System.out.println("Barber numHairCuts       : " + barberNumCuts.get());
 System.out.println("Barber sleeps            : " + BlockingQueueSleepingBarber.barberSleeps.get());
 System.out.println("Customer on cuttingChair : " + customerCuttingChair.get());
 System.out.println("Customer on waitingChair : " + customerWaitingChair.get());
 System.out.println("Customer left without cut: " + customerLeft.get());
  }
}
```