Soluzione esercizio 1

La race condition presente nel programma è di tipo read-modify-write ed è dovuta alle operazioni di incremento eseguite per le variabili *entrate, uscite* e *pedaggio* dello stato condiviso. Inoltre, le variabili *uscite* e *pedaggio* sono interdipendenti, quindi i due incrementi formano una compound action da eseguire in maniera atomica.

Soluzione con synchronized block

In questa soluzione va introdotto un blocco synchronized attorno alla porzione di codice che deve essere atomica, sfruttando come oggetto di sincronizzazione l'istanza dello stato condiviso: *autostrada*. L'operazione di *percorriAutostrada* fra le due operazioni non necessita di protezione, visto che non accede allo stato condiviso.

```
class Automobilista {
...
  @Override
  public void run() {
    System.out.println("Automobilista " + id + ": partito");

    for (int i = 0; i < 500; i++) {
      vaiVersoAutostrada();

      synchronized (autostrada) {
        autostrada.entrate++;
      }
      percorriAutostrada();
      final int pedaggioTratta = ThreadLocalRandom.current().nextInt(10, 20);

      synchronized (autostrada) {
        autostrada.uscite++;
        autostrada.pedaggi += pedaggioTratta;
      }
      pedaggiPagati += pedaggioTratta;
    }
    System.out.println("Automobilista " + id + ": terminato");
  }
```

Soluzione con metodi synchronized

Per risolvere il problema con i metodi synchronized, bisogna estrarre la logica dei due blocchi synchronized nei due nuovi metodi della classe *Autostrada*: *entra* e *esci*. I metodi vanno dichiarati come synchronized in modo da usare l'istanza della classe come oggetto di sincronizzazione, esattamente come veniva fatto per la soluzione precedente. In seguito, la logica all'interno del metodo run della classe *Automobilista* può essere semplificata, sostituendola con l'invocazione dei due metodi. Infine, le variabili condivise *entrate*, *uscite* e *pedaggi* possono essere rese private, visto che l'accesso avviene esclusivamente attraverso i metodi d'istanza *entra* e *esci*.

```
class Automobilista {
...
  @Override
  public void run() {
    System.out.println("Automobilista " + id + ": partito");

    for (int i = 0; i < 500; i++) {
      vaiVersoAutostrada();

      autostrada.entra();

      percorriAutostrada();

      final int pedaggioTratta = ThreadLocalRandom.current().nextInt(10, 20);
      autostrada.esci(pedaggioTratta);
      pedaggiPagati += pedaggioTratta;
```

```
    }
    System.out.println("Automobilista " + id + ": terminato");
  }
}

class Autostrada {
  private int entrate = 0;
  private int uscite = 0;
  private int pedaggi = 0;

  public synchronized void entra() {
    entrate++;
  }

  public synchronized void esci(final int pedaggio) {
    this.pedaggi += pedaggio;
    this.uscite++;
  }


  public synchronized int getEntrate() {
    return entrate;
  }

  public synchronized int getUscite() {
    return uscite;
  }

  public synchronized int getPedaggi() {
    return pedaggi;
  }
}
```

Soluzione con explicit locks

Per questa soluzione va introdotto un ReentrantLock, che per convenienza può essere dichiarato direttamente nella classe *Autostrada*, in modo da legarlo in maniera esplicita allo stato condiviso. Successivamente, come per la soluzione con i blocchi synchronized, vanno protette le compound actions, racchiudendole tra le chiamate ai metodi lock() ed unlock(). Inoltre, è buona prassi racchiudere la porzione di codice da proteggere in un blocco try / finally, in modo da garantire che il lock venga sempre rilasciato, in particolare nel caso di lancio d'eccezioni.

```
class Autostrada {
  public final Lock lock = new ReentrantLock();
  public int entrate = 0;
  public int uscite = 0;
  public int pedaggio = 0;
}

class Automobilista {
...
  @Override
  public void run() {
    System.out.println("Automobilista " + id + ": partito");

    for (int i = 0; i < 500; i++) {
      vaiVersoAutostrada();
      autostrada.lock.lock();
      try {
        autostrada.entrate++;
      } finally {
        autostrada.lock.unlock();
      }
      percorriAutostrada();
      final int pedaggioTratta = ThreadLocalRandom.current().nextInt(10, 20);

      autostrada.lock.lock();
      try {
        autostrada.uscite++;
        autostrada.pedaggio += pedaggioTratta;
      } finally {
        autostrada.lock.unlock();
```

```
      }
      pedaggiPagati += pedaggioTratta;
    }
    System.out.println("Automobilista " + id + ": terminato");
  }
}
```

Soluzione con metodi ed explicit locks

Per questa soluzione si procede come per la soluzione con i metodi synchronized. Al posto di dichiarare i metodi come synchronized, si aggiunge alla classe *Autostrada* un'istanza di ReentrantLock e si usa tale lock per proteggere le compound actions nei metodi *entra* e *esci*. Quindi, la porzione di codice lato *Automobilista* resta identica alla versione con synchronized methods. Il codice che accede allo stato condiviso è completamente incapsulato e protetto all'interno della classe *Autostrada*.

```
class Autostrada {
  private final Lock lock = new ReentrantLock();
  private int entrate = 0;
  private int uscite = 0;
  private int pedaggio = 0;

  public void entra() {
    lock.lock();
    try {
      entrate++;
    } finally {
      lock.unlock();
    }
  }

  public void esci(final int pedaggio) {
    lock.lock();
    try {
      this.pedaggio += pedaggio;
      this.uscite++;
    } finally {
      lock.unlock();
    }
  }

  public int getEntrate() {
    lock.lock();
    try {
      return entrate;
    } finally {
      lock.unlock();
    }
  }

  public int getUscite() {
    lock.lock();
    try {
      return uscite;
    } finally {
      lock.unlock();
    }
  }

  public int getPedaggio() {
    lock.lock();
    try {
      return pedaggio;
    } finally {
      lock.unlock();
    }
  }
}
```

Soluzione esercizio 2

Il programma soffre di una race condition di tipo check-then-act: viene presa una decisione sulla base di valori condivisi non protetti (*occupati, disponibili*) che possono essere modificati concorrentemente fra l'"if" della decisione e l'operazione che segue. La correttezza dell'esecuzione dell'operazione può quindi venire compromessa.

Anche gli incrementi della variabile *totUtilizzati* così come gli incrementi e decrementi della variabile *occupati* fanno parte dello stato condiviso e soffrono della race condition read-modify-write. Vanno perciò protette usando il medesimo lock.

Infine bisogna assicurarsi che anche le letture alle variabili *totUtilizzi* e *totOccupati* attraverso i rispettivi metodi public vengano protette da lock.

Soluzione con synchronized block

Il problema può essere risolto utilizzando un intrinsic lock (parola riservata synchronized) sfruttando l'istanza dell'oggetto come lock per proteggere gli accessi alle variabili condivise (visto che lo stato condiviso è composto da variabile d' istanza). Di conseguenza si rende atomica la compound action che verifica lo stato ed esegue le operazioni successive.

```
public boolean occupa() {
  // Verifica disponibilita bagni liberi!
  synchronized (this){
    if (occupati < disponibili) {
      // Bagno libero! Occupa
      occupati++;
      totUtilizzi++;
    } else {
      // Tutti i bagni sono occupati!
      totOccupati++;
      return false;
    }
  }

  // Utilizza il bagno
  try {
    Thread.sleep(ThreadLocalRandom.current().nextLong(5, 15));
  } catch (InterruptedException e) {
    e.printStackTrace();
  }

  // Libera il bagno
  synchronized (this){
    occupati--;
  }
  return true;
}

public int getTotaleUtilizzo() {
  synchronized (this){
    return totUtilizzi;
  }
}

public int getTotaleOccupato() {
  synchronized (this){
    return totOccupati;
  }
}
```

Soluzione con explicit locks

Per risolvere l'esercizio con gli explicit locks è innanzitutto necessario istanziare un nuovo ReentrantLock. Bisogna poi proteggere le compound actions usando le chiamate a lock e unlock avendo cura di racchiudere il codice da proteggere in un blocco try / finally.

```
private ReentrantLock lock = new ReentrantLock();
...
public boolean occupa() {
  lock.lock();
  try {
    if (occupati < disponibili) {
      // Bagno libero! Occupa
      occupati++;
      totUtilizzi++;
    } else {
      // Tutti i bagni sono occupati!
      totOccupati++;
      return false;
    }
  } finally {
    lock.unlock();
  }

  // Utilizza il bagno
  try {
    Thread.sleep(ThreadLocalRandom.current().nextLong(5, 15));
  } catch (InterruptedException e) {
    e.printStackTrace();
  }

  lock.lock();
  try {
    occupati--;
  } finally {
    lock.unlock();
  }
  return true;
}

public int getTotaleUtilizzo() {
  lock.lock();
  try {
    return totUtilizzi;
  } finally {
    lock.unlock();
  }
}

public int getTotaleOccupato() {
  lock.lock();
  try {
    return totOccupati;
  } finally {
    lock.unlock();
  }
}
```

Soluzione con metodi synchronized

Per risolvere il problema con i metodi synchronized è necessario rifattorizzare il codice estraendo la logica della compound action e di quella per liberare il bagno nei metodi *provaOccupare* e *libera.* Il metodo *occupa* infine, non contenendo più alcun codice necessario a garantire la protezione dello stato condiviso, risulta essere più leggibile.

```
public boolean occupa() {
  if (provaOccupare() == false) {
    return false;
  }
  // Utilizza il bagno
```

```
    try {
      Thread.sleep(ThreadLocalRandom.current().nextLong(5, 15));
    } catch (InterruptedException e) {
      e.printStackTrace();
    }

    libera();
    return true;
  }

  private synchronized boolean provaOccupare() {
    if (occupati < disponibili) {
      // Bagno libero! Occupa
      occupati++;
      totUtilizzi++;
    } else {
      // Tutti i bagni sono occupati!
      totOccupati++;
      return false;
    }
    return true;
  }

  private synchronized void libera() {
    occupati--;
  }

  public synchronized int getTotaleUtilizzo() {
    return totUtilizzi;
  }

  public synchronized int getTotaleOccupato() {
    return totOccupati;
  }
```

Soluzione con explicit locks e metodi

In alternativa ai metodi synchronized si possono dichiarare gli stessi metodi ma proteggendo le compound actions usando gli explicit locks nel seguente modo.

```
  private ReentrantLock lock = new ReentrantLock();
  ...
    public boolean occupa() {
    if (provaOccupare() == false) {
      return false;
    }
    // Utilizza il bagno
    try {
      Thread.sleep(ThreadLocalRandom.current().nextLong(5, 15));
    } catch (InterruptedException e) {
      e.printStackTrace();
    }

    libera();
    return true;
  }

  private boolean provaOccupare() {
    lock.lock();
    try {
      if (occupati < disponibili) {
        // Bagno libero! Occupa
        occupati++;
        totUtilizzi++;
      } else {
        // Tutti i bagni sono occupati!
        totOccupati++;
        return false;
      }
      return true;
    } finally {
      lock.unlock();
    }
```

```
  }

  private void libera() {
    lock.lock();
    try {
      occupati--;
    } finally {
      lock.unlock();
    }
  }

  public int getTotaleUtilizzo() {
    lock.lock();
    try {
      return totUtilizzi;
    } finally {
      lock.unlock();
    }
  }

  public int getTotaleOccupato() {
    lock.lock();
    try {
      return totOccupati;
    } finally {
      lock.unlock();
    }
  }
```

Soluzione esercizio 3

```
import java.util.ArrayList;
import java.util.Random;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

class Account {
  public final Lock lock;
  public long accountBalance;

  public Account(final int initialAmount) {
    accountBalance = initialAmount;
    lock = new ReentrantLock();
  }
}

class User implements Runnable {
  private final int ID;
  private final int delay;
  private long wallet;
  private final Random random;

  private final Account account;

  public User(final Account bankAccount, final int id, final int delay) {
    this.account = bankAccount;
    this.ID = id;
    this.delay = delay;
    this.wallet = 0;
    this.random = new Random();
    log("creato. Preleva ogni " + delay + " ms");
  }

  @Override
  public void run() {
    boolean isRunning = true;

    while (isRunning) {
      try {
        // Attendi prima di proseguire
        Thread.sleep(delay);
      } catch (final InterruptedException ex) {
        return;
      }
```

```
      // Calcola l'importo da prelevare [5, 50]
      final long requestedAmount = random.nextInt(46) + 5;
      final long withdrawnAmount;
      final long startBalance;

      // Richiedi accesso esclusivo al conto corrente
      account.lock.lock();
      try {
        startBalance = account.accountBalance;

        if (account.accountBalance == 0) {
          withdrawnAmount = 0;
        } else if (account.accountBalance < requestedAmount) {
          withdrawnAmount = account.accountBalance;
          // Preleva quello che rimane sul conto
          account.accountBalance = 0;
        } else {
          withdrawnAmount = requestedAmount;
          // Aggiorna conto
          account.accountBalance -= requestedAmount;
        }
      } finally {
        // Rilascia lock
        account.lock.unlock();
      }

      // Aggiorna il proprio portafoglio con la somma prelevata
      wallet += withdrawnAmount;

      // Scrivi a schermo il messaggio evitando di mantenere il lock
      if (withdrawnAmount == 0) {
        log("conto corrente vuoto!");

        // Termina esecuzione
        isRunning = false;
      } else if (withdrawnAmount < requestedAmount) {
        log("sono riuscito a prelevare solo " + withdrawnAmount
            + "$ dal conto contenente invece di " + requestedAmount
            + "$");

        // Termina esecuzione
        isRunning = false;
      } else
        // Notifica l'esito positivo
        log("prelevo " + requestedAmount + "$ dal conto contenente "
            + startBalance + "$. Nuovo saldo "
            + (startBalance - requestedAmount) + "$");
    }
  }

  public long getWallet() {
    return wallet;
  }

  public void log(final String message) {
    System.out.println("Utente " + ID + ": " + message);
  }
}

public class S2Esercizio3 {

  static final int INITIAL_AMOUNT = 10000;
  static final int USERS = 5;

  public static void main(final String[] x) {
    final Random random = new Random();
    final Account account = new Account(INITIAL_AMOUNT);
    // Preparo due nuovi oggetti di tipo Race

    final ArrayList<User> users = new ArrayList<User>();
    final ArrayList<Thread> allUserThread = new ArrayList<Thread>();
    for (int i = 0; i < USERS; i++) {
      // Genera numero casuale tra 5 e 20
      final int delay = 5 + random.nextInt(16);
      final User curUser = new User(account, i, delay);
      users.add(curUser);
```

```
      allUserThread.add(new Thread(curUser));
    }
    System.out.println("--------------------");

    try {
      // Avvia tutte le threads
      for (final Thread t : allUserThread)
        t.start();

      // Resta in attesa che tutte le threads abbiano terminato
      for (final Thread t : allUserThread)
        t.join();
    } catch (final InterruptedException e) {
      // Nessuna gestione delle eccezioni
    }
    System.out.println("--------------------");
    long totalUserCash = 0;
    for (final User u : users) {
      final long userWallet = u.getWallet();
      totalUserCash += userWallet;
      u.log("ha prelevato : " + userWallet + "$");
    }
    System.out.println("Totale soldi prelevati : " + totalUserCash + "$");
    final long balance = account.accountBalance;
    System.out.println("Bilancio finale conto corrente :" + balance + "$");
    System.out.println("Totale prelievi e saldo finale : " + (totalUserCash + balance) + "$");
    System.out.println("Somma iniziale conto corrente  : " + INITIAL_AMOUNT + "$");
    System.out.println("--------------------");
    System.out.println("Simulation finished.");
  }
}
```