אוניברסיטת בן-גוריון

מדור בחינות

מספר נבחן:_____

רשמו תשובותיכם בגיליון התשובות בלבד
תשובות מחוץ לגיליון לא יבדקו.

בהצלחה!

תאריך הבחינה: 1.2.2010
שם המורה: ד"ר מיכאל אלחדד
ד"ר מני אדלר
מר אוריאל ברגיג
שם הקורס: תכנות מערכות
מספר הקורס: 202-1-2031
מיועד לתלמידי: מדעי המחשב, הנדסת תוכנה
שנה: תש"ע
סמסטר: א'
מועד: א'
משך הבחינה: שלש שעות
חומר עזר: אסור

שאלה 1 (30 נקודות)

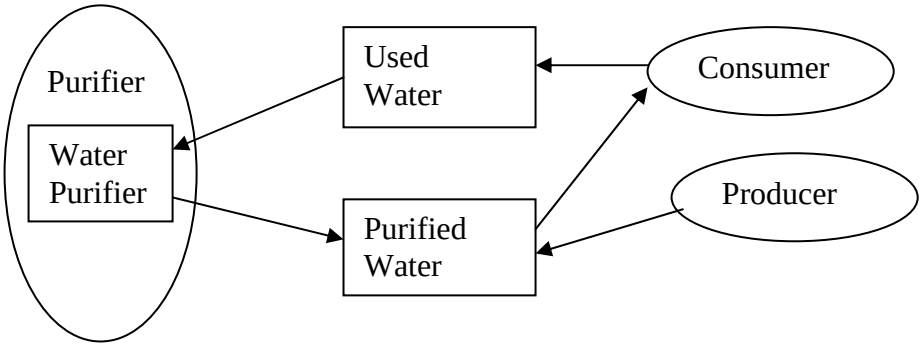
בענף הבניה הירוקה, הולך ופושט המנהג להתקין מערכת 'מים אפורים' לשימוש חוזר במי המקלחת.
בצנרת הרגילה, המים הזורמים מהברז מתנקזים למערכת הביוב. במערכת 'מים אפורים' מטוהרים מים משומשים אלו ומוחזרים לשימוש חוזר.
בשאלה זו, נציג סימולציה פשוטה של המערכת.

המערכת מורכבת מהאובייקטים הפסיביים הבאים:

- שני מיכלים (**WaterPool**) - עבור מים מטוהרים ועבור מים משומשים - המממשים את הממשק **Pool**.
- מטהר מים (**WaterPurifier**) המממש את הממשק **Purifier**. בממשק מוגדרת המתודה **getQuality()** המחזירה את כמות המים באחוזים שניתן לטהר באיכות טובה והמתודה **purify** המקבלת כמות של מים משומשים ומחזירה כמות של מים מטוהרים.

בסימולציה שלנו, רצים 3 אובייקטים אקטיביים:

- **Producer**: ת'רד הרץ מעל המשימה **ProducingTask**: הוספת מים למיכל המים המטוהרים (למשל זרימת המים מהקו העירוני לצנרת הבית).
- **Consumer**: ת'רד הרץ מעל המשימה **ConsumingTask**: צריכת מים ממיכל המטהורים והעברתם בתום השימוש למיכל המים המשומשים (למשל צרכן הפותח ברז, צורך מים, והמים מתנקזים למאגר המים המיועדים לטיהור).
- **Purifier**: ת'רד הרץ מעל המשימה **PurifyungTask**: לקיחת מים ממיכל המים המשומשים, טיהורם בעזרת מטהר המים, והוספתם למיכל המים המטוהרים (כמו פעולת המערכת לטיהור המים האפורים).



התבוננו בקטעי הקוד המממשים את המערכת וענו על השאלות שאחר כך:

```
interface Pool {
  long getCapacity();
  float getContent();
  float remove();
  void add(float addition);
}

class WaterPool implements Pool {
  private final long _capacity;
  private float _content;

  WaterPool(float content, long capacity)  { _content = content; _capacity = capacity; }

  public long getCapacity() { return _capacity; }
  public synchronized float getContent () { return _content; }
  public synchronized float remove() {
    while (_content == 0)
      try { wait(); } catch (InterruptedException e) { return 0; }
    float ret = _content;
    _content = 0;
    notifyAll();
    return ret;
  }

  public synchronized void add(float addition) {
    while (_content + addition  > _capacity)
      try { wait(); } catch (InterruptedException e) { return; }
     _content += addition;
    notifyAll();
  }
}
```

```
interface Purifier {
  float getQuality();
  float purify(float content);
}

class WaterPurifier implements Purifier {
  private final float _quality;
  WaterPurifier(float quality) { _quality = quality; }
  public float getQuality() { return _quality; }
  public float purify (float content) {
```

```
    try {
        Thread.sleep((long)content);        // sleep to simulate using the water
        } catch (InterruptedException e) { return 0; }
    return _quality * content;
  }
}
```

```
class ProducingTask implements Runnable {
  private Pool _purifiedPool;
  ProducingTask (Pool purifiedPool) {
    _purifiedPool = purifiedPool;
  }
  public void run() {
    while (true)
      _purifiedPool.add(_purifiedPool.getCapacity());
  }
}
```

```
class PurifyingTask implements Runnable {
  private Pool _usedPool, _purifiedPool;
  private Purifier _purifier;
  PurifyingTask (Pool usedPool, Pool purifiedPool, Purifier purifier) {
    _usedPool = usedPool; _purifiedPool = purifiedPool; _purifier = purifier;
  }
  public void run() {
    while (true)
      _purifiedPool.add(_purifier.purify(_usedPool.remove()));
  }
}
```

```
class ConsumingTask implements Runnable {
  private Pool _purifiedPool, _usedPool;
  ConsumingTask (Pool purifiedPool, Pool usedPool) {  _purifiedPool = purifiedPool; _usedPool = usedPool; }
  public void run() {
    while (true) {
      float w = _purifiedPool.remove();
      try { Thread.sleep((long)w); } catch (InterruptedException e) { return; }
      _usedPool.add(w);
    }
  }
}
```

```
class Test {
  public static void main(String[] args) {
    Pool purifiedWaterPool = new WaterPool(0, 1000);
    Pool usedWaterPool = new WaterPool(0, 1000);
    Purifier waterPurifier = new WaterPurifier(0.8);
    new Thread(new ProducingTask(purifiedWaterPool)).start();
    new Thread(new PurifyingTask(usedWaterPool, purifiedWaterPool, waterPurifier)).start();
    new Thread(new ConsumingTask(purifiedWaterPool, usedWaterPool)).start();
  }
}
```

א. הגדירו תכונה נשמרת (invariant) ותנאי התחלה וסיום (pre/post conditions) עבור הממשק Pool. במידה ונדרש עיצוב מחדש של הממשק (refactoring) על פי העקרונות שנלמדו בהרצאה (Test Driven Design), בצעו זאת.
[15 נקודות]

ב. האם המערכת בטוחה תחת חישוב מקבילי? נמקו. [5 נקודות]

ג. הראו כיצד עלולה המערכת להגיע לחבק (deadlock), ותקנו את הקוד כך שהחבק ימנע [10 נקודות]

שאלה 2 (30 נקודות)

נתונות המחלקות הבאות המייצגות מחסנית של אירועים. הקוד תקין:

```
class Event {
public:
  Event(int id) : _id(id) {};
  virtual int id() {return _id;}
  virtual void print() { cout << "Event id is " << id() << endl; }
private:
  const int _id;
};

class EventNode {
public:
  EventNode(){};
  Event *data;
  EventNode *next;
};

class EventsStack {
  EventNode *_top;
public:
```

```
  EventsStack();
  void push(Event *data);
  void pop();
  Event* top() const;
};

EventsStack::EventsStack() : _top(0) { } ;
void EventsStack::push(Event *e) {
  EventNode *q = new EventNode();
  q->data = e;
  q->next = _top;
  _top = q;
}
void EventsStack::pop(){
  if ( _top == 0 ) return;
    EventNode *oldTop = _top;
  _top = _top->next;
  delete oldTop;
}
Event* EventsStack::top() const {
  return _top->data;
}
```

בסעיפים א-ג יש לצייר את תמונת הזיכרון של התהליך הכוללת ערכים הנמצאים במחסנית, ערכים הנמצאים בערימה ומצביעי vtable אם יש. אין צורך לאייר את ה vtables. גודלם של מצביעים ו- int הוא 4. נקבע כי כתובות בערימה הם בתחום 1000 עד 3000 וכי כתובת המחסנית מתחילה ב 5000 ועולה כשהמחסנית גדלה.
כל סעיף יש לאייר מחדש. לכל תא זיכרון באיור יש לכלול מספר המייצג את כתובתו.

א. ציירו את תמונת הזיכרון לאחר ביצוע השורות הבאות [2 נקודות]

```
void main() {
  EventsStack eventStack;
```

ב. ציירו את תמונת הזיכרון לאחר ביצוע השורות הבאות [6 נקודות]

```
void main() {
  EventsStack eventStack;
  Event e1(1);
  eventStack.push(&e1);
  Event e2(2);
  eventStack.push(&e2);
```

נוספו 2 מחלקות חדשות:

```
class BuyingEvent : public Event {
```

```
public:
  BuyingEvent(int id) : Event(id) {};
};

class SellingEvent : public Event {
public:
  SellingEvent(int id, int price) : Event(id), _price(price) {};
  int _price;
};
```

ג. ציירו את תמונת הזיכרון לאחר ביצוע השורות הבאות [6 נקודות]

```
void main() {
  EventsStack eventStack;
  Event *e1 = new BuyingEvent(1);
  eventStack.push(e1);
  Event *e2 = new SellingEvent(2, 20);
  eventStack.push(e2);
```

ד. בעת הרצת התוכנית הבאה התגלתה דליפת זיכרון. יש לפתור את דליפת הזיכרון ע"י עדכון המחלקה EventsStack וללא שינוי ה main הנתון: [6 נקודות]

```
void main() {
  EventsStack eventStack;
  Event e1(1);
  eventStack.push(&e1);
}
```

ה. ביצוע השורות הבאות אינו גורם לשכפול המחסנית כולה.
הוסיפו אופרטור ההשמה למחלקה EventStack כך שיבצע שכפול המחסנית כולה (העתקה עמוקה). יש לשכפל את הצמתים אך לא את הנתונים כך שלאחר ביצוע השמה כמודגם מטה, יהיו הצבעות משתי מחסניות שונות על על אותם e1,e2.
כמו כן יש לאייר את תמונת הזיכרון שתתקבל מביצוע השורות הבאות לאחר הוספת אופרטור ההשמה.
[10 נקודות]

```
void main() {
  EventsStack eventStack;
  Event e1(1);
  eventStack.push(&e1);
  Event e2(2);
  eventStack.push(&e2);

  EventsStack eventStack1 = eventStack;
```

שאלה 3 (30 נקודות)

בשני התרגילים האחרונים, עסקנו בפרוטוקול ה STOMP התומך בפרסום של הודעה לתור של מנויים.
בשאלה זו, נעסוק בגרסה פשוטה יותר של פרוטוקול זה.

התחברות לשרת נעשית על ידי שליחת ההודעה COONECT לשרת ה STOMP

```
CONNECT
login: <username>
passcode:<passcode>

^@
```

לשם פשטות השאלה, בגירסה זו של הפרוטוקול, השרת <u>אינו</u> מחזיר הודעה CONNECTED עם sessionId, כפי שמימשתם בתרגיל.

הצטרפות כמנוי לתור נעשית על ידי שליחת ההודעה SUBSCRIBE לשרת ה STOMP:

```
SUBSCRIBE
destination: <queue name>

^@
```

ביטול רישום לתור ניתן על ידי שליחת ההודעה UNSUBSCRIBE לשרת ה STOMP:

```
UNSUBSCRIBE
destination: <queue name>

^@
```

שליחת הודעה לקבוצת מנויים רשומים, ניתנת על ידי שליחת הודעת SEND ל שרת ה STOMP המכילה את שם קבוצת המנויים, ואת תוכן ההודעה למשלוח.

```
SEND
destination: <queue name>

<message string>
^@
```

כתגובה, ישלח השרת הודעת MESSAGE לכל אחד מהלקוחות שנרשמו לתור

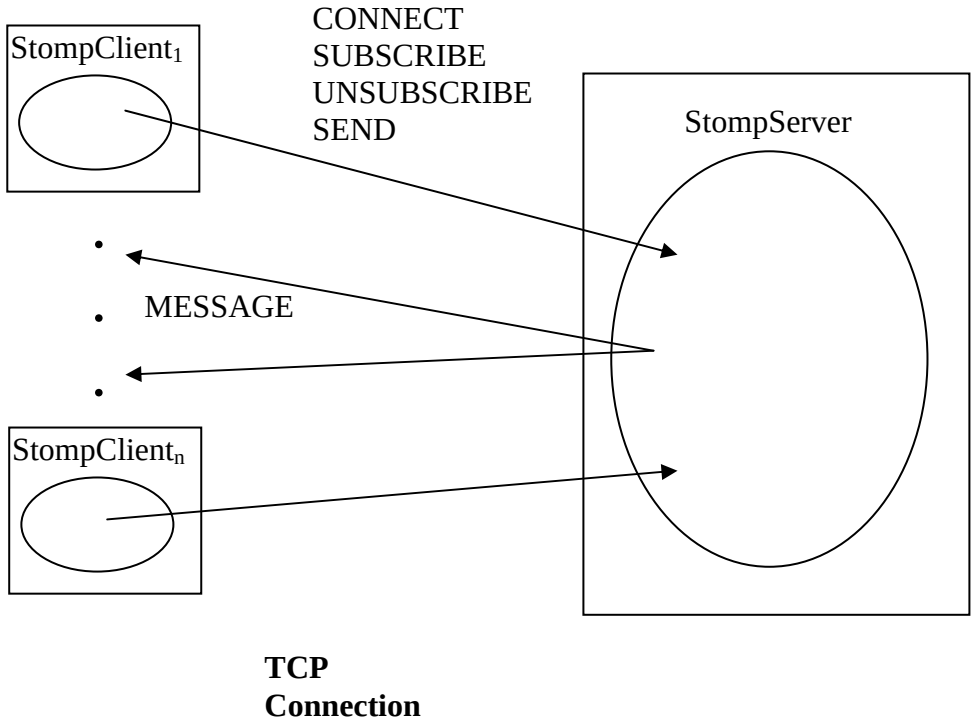
```
MESSAGE
destination: <queue name>
message-id: <message-identifier>

<message string>
^@
```

על פי הפרוטוקול, על כל אחת מבקשות הלקוח (CONNECT, SUBSCRIBE, UNSUNSCRIBE, SEND) עשוי השרת להחזיר הודעת שגיאה (ERROR).
בשאלה זו נניח כי הבקשות תמיד מצליחות, כך שלא נשלחות אף פעם הודעות שגיאה.
ההודעות היחידות שנשלחות מהשרת הינן הודעות מסוג MESSAGE.

באופן זה יכולים לקוחות שונים להירשם לתורים שונים, לשלוח הודעות לתורים, ולקבל הודעות שנשלחו לתורים אליהם הן נרשמו.

נניח כי קיים מימוש של שרת STOMP המקבל ושולח הודעות ב TCP – כמו זה שמימשתם בתרגיל 3 ו 4.



במודל זה, ה StompClient מגדיר TCP Socket דרכו הוא שולח ומקבל הודעות, בפורמט של STOMP אשר תואר לעיל.

כדי לפשט את הגישה לשרת הוחלט להגדיר תהליך ביניים בשם StompClientRMI, המכיל Remote Object (RMI) הממש ממשק התחברות לשרת ה Stomp

```
public interface StompConnector extends java.rmi.Remote {
   StompOperator connect(String login, String passcode)  throws java.rmi.RemoteException,IOException;
}
```

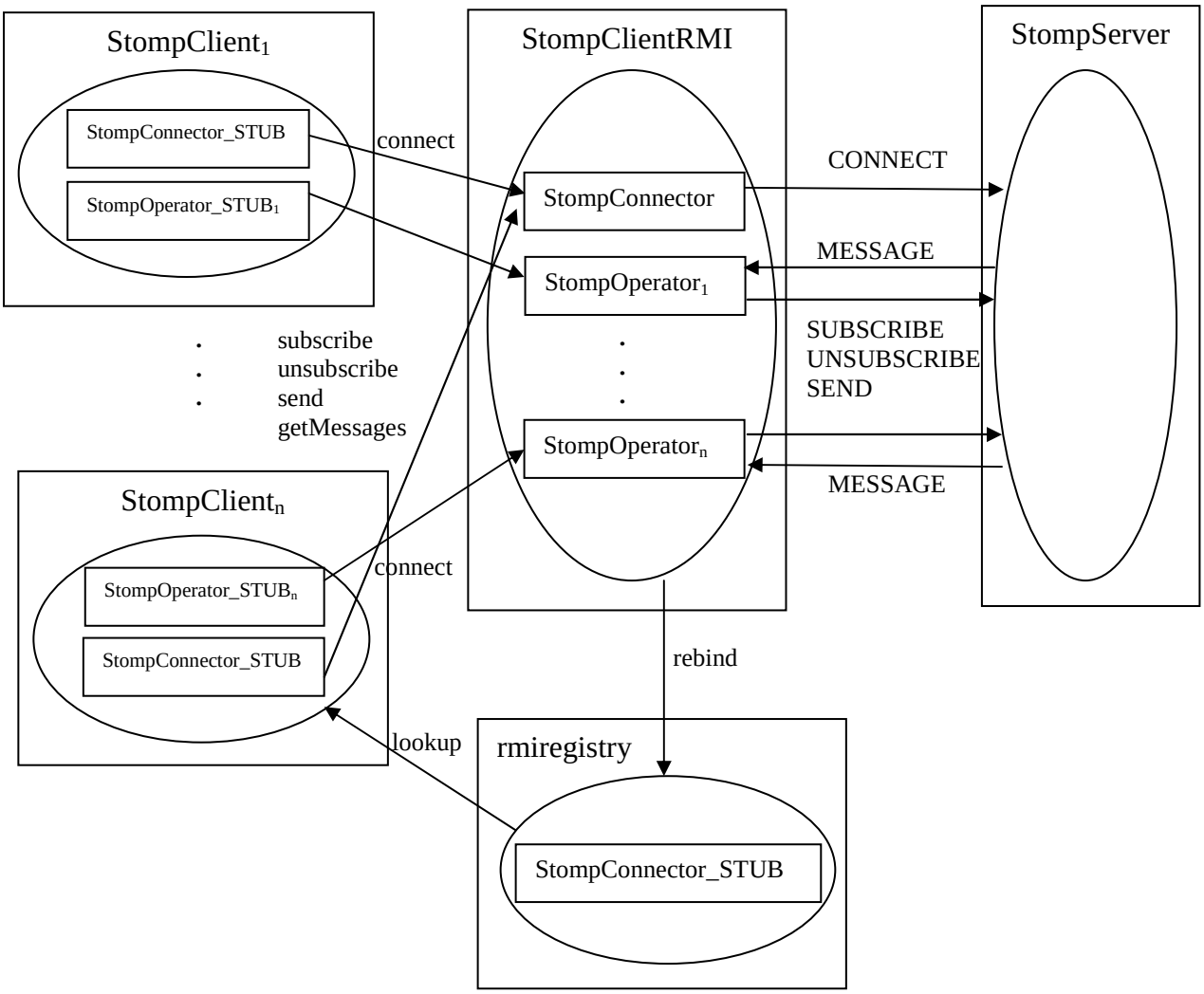
המתודה connect מחברת את הלקוח לשרת ה Stomp על ידי שליחת ההודעה CONNECT (כזכור, אני מניחים כי הפעולה תמיד מצליחה. וכן אין הודעת CONNECTED מהשרת).
המתודה מחזירה RemoteObject הממש ממשק לפעולות הלקוח מול שרת ה Stomp, על פי הפרוטוקול:

```
public interface StompOperator extends java.rmi.Remote {
   void subscribe(String groupName) throws java.rmi.RemoteException,IOException;
   void unsubscribe(String groupName) throws java.rmi.RemoteException,IOException;
   void send(String groupName, String str) throws java.rmi.RemoteException,IOException;
   List<String> getMessages() throws java.rmi.RemoteException,IOException;
}
```

המתודות **subscribe, unsubscribe, send** שולחות לשרת את ההודעות SUBSCRIBE, UNSUBSCRIBE, SEND. המתודה **getMessages** מחזירה את תוכן ההודעות שהגיעו מהשרת (ע"י פעולת MESSAGE) ועדיין לא נקראו על ידי הלקוח.

ה **StompClientRMI** מתווך בין בין ה **StompClient** ל **StompServer**, באופן הבא:

- הוא מגדיר אובייקט הממש את **StompConnector** ומפרסמו ב **rmiregistry**.
- ה **StompClient** ניגש ל **rmiregistry** ומקבל את ה **StompConnector** כך שהוא יכול לבצע את פעולת ההתחברות, ולקבל כערך מוחזר אובייקט הממש את **StompOperator** לשם ביצוע פעולות באופן נוח מול שרת ה Stomp.



RMI Connection **TCP Connection**

להלן מימוש של שני הממשקים (ללא מימוש המתודה **getMessages()** של **StompOperator**).

```
public class StompConnectorImpl extends java.rmi.server.UnicastRemoteObject implements StompConnector {
```

```
  protected String _stompServerHost;
  protected int _stompServerPort;

  public StompConnectorImpl(String stompServerHost, int stompServerPort)  throws java.rmi.RemoteException {
    _stompServerHost = stompServerHost;
    _stompServerPort = stompServerPort;
  }

  public StompOperator connect (String login, String passcode) throws java.rmi.RemoteException,IOException {
    Socket socket = new Socket(_stompServerHost, _stompServerPort);
    PrintWriter out = new PrintWriter(new OutputStreamWriter(socket.getOutputStream(),"UTF-8"));
    out.print("CONNECT\nlogin: " + login + "\npasscode: " + passcode + "\n\n" + '\0');
    return new StompOperatorImpl(socket.getInputStream(),socket.getOutputStream());
  }
}
```

```
public class StompOperatorImpl
  extends java.rmi.server.UnicastRemoteObject
  implements StompOperator
 {
  protected PrintWriter _writer;
  protected BufferedReader _reader;

  public StompOperatorImpl(InputStream in, OutputStream out) throws java.rmi.RemoteException, IOException {
    _reader = new BufferedReader(new InputStreamReader(in,"UTF-8"));
    _writer = new PrintWriter(new OutputStreamWriter(out,"UTF-8"));
  }

  public void subscribe (String group)   throws java.rmi.RemoteException, IOException {
      synchronized (_writer) { _writer.print("SUBSCRIBE\ndestination: " + group + "\n\n" + '\0'); }
  }

  public void unsubscribe (String group)  throws java.rmi.RemoteException, IOException {
      synchronized (_writer) { _writer.print("UNSUBSCRIBE\ndestination: " + group+ "\n\n" + '\0'); }
  }

  public void send (String group, String str)   throws java.rmi.RemoteException, IOException {
      synchronized (_writer) { _writer.print("SEND\ndestination: " + group+ "\n\n" + str + "\n" + '\0'); }
  }

  public List<String> getMessages() throws java.rmi.RemoteException {
  …
  }
}
```

במימוש שלכם לפרוטוקול בתרגילים 3,4, האזין ה StompClient על ה Socket ל StompServer, כדי לחלץ את ההודעות שנשלחו על ידי השרת, כאשר הן מתקבלות, ללא בקשה מיוחדת לכך מצד המשתמש מעבר לרישומו הראשוני בתור.
בשאלה זו, לעומת זאת, משיכה של הודעות דרך הממשק StompOperator, נעשית באופן יזום, רק בעקבות הפעלת המתודה getMessages() – כלומר לא מספיק להרשם לתור, צריך גם לבקש לחלץ את ההודעות מידי פעם.

א. אחד הסטודנטים, בקורס משנה שעברה, הציע לממש את המתודה getMessages() כקריאה של המידע שהתקבל דרך השדה reader_ ב StompOperatorImpl.
הסבירו מדוע מימוש שכזה עשוי להביא את ת'רד הנוכחי (של RMI) המריץ את המתודה getMessages() למצב blocked, אף שהוא אינו נעול על ידי ת'רד אחר. [2 נקודות]

ב. עדכנו את המחלקה StompOperatorImpl, וממשו את המתודה getMessages(), כך שתימנע הבעיה בסעיף א' (על ידי הגדרת ת'רד אחר למשימת הקריאה מה reader_). [10 נקודות]

ג. השלימו את התוכנית StompClientRMI. [3 נקודות]
הניחו כי תהליך של rmiregistry רץ ב host שכתובתו "132.23.5.8", על port 2010

```
public class StompClientRMI {
  public static void main(String[] args) {
    try {
      //@TODO
    }
    catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

ד. השלימו את מתודת ה main של התוכנית StompClient, כך
- קבלת StompConnector
- התחברות לשרת ה Stomp
- רישום לתורים ששמם q1, q2
- שליחת ההודעה "Suzy Surprise" לתור q3
- המתנה למשך דקה
- הדפסה על המסך של ההודעות שנשלחו לתורים q1, q2
- ביטול הרישום לתורים q1, q2
[6 נקודות]

```
public class StompClient {
  public static void main(String[] args) {
    try {
        //@TODO1: get  StompConnector
       //@TODO2: connect to the stomp server
       //@TODO3: subscribe to queues q1,q2
       //@TODO4: send  "Suzy Se" message to queue q3
      Thread.sleep(60000);
```

```
        //@TODO5: print the messages that were sent to q1 and q2
       //@TODO6: unsubscribe  queues q1,q2
    }
   catch (Exception e) {
     e.printStackTrace();
   }
  }
}
```

ה. כמה פעולות תקשורת נדרשות עבור שתי הפעולות הראשונות ב **main** של **StompClient** (**@TODO1,** **@TODO2**)? פעולת תקשורת מוגדרת כהעברת מידע בפרוטוקול TCP מתהליך לתהליך, בכיוון אחד. [6 נקודות]

ו. בתרגילים 3, 4 כתבתם שני מימושים ל StompServer, הנבדלים במודל התקשורת עם הלקוחות:

- Multi-Client Server
- The Reactor

האם מספר פעולות התקשורת שתיארתם בסעיף ד' תלוי במודל התקשורת הנבחר Multi-client /Reactor Server? [3 נקודות]

שאלה 4 (10 נקודות)

לוח זמני הטיסות של נמל התעופה בן גוריון מאוחסן במערכת לניהול בסיסי נתונים.
לכל טיסה יש מספר זיהוי יחודי. בנוסף, מתאפיינת הטיסה על פי היעד שלה, מספר הטרמינל, שער היציאה, תאריך ושעת הטיסה, והמטוס המבצע את הטיסה.
מטוס מתאפיין על פי מספר יחודי, וציון הסוג שלו.
לכל סוג מטוס יש מחרוזת המתארת את המפרט הטכני שלו, וכן שדה המציין את קיבלת הנוסעים המקסימאלית.

א. הגדירו מודל נתונים (טבלאות ומפתחות) עבור המערכת שתוארה לעיל. [5 נקודות]
ב. הגדירו שאילתת SQL המחזירה את הקיבולת המקסימאלית של המטוסים היוצאים למדריד בתאריך 1.2.10. התוצאות צריכות להיות ממוינות בסדר יורד. [5 נקודות]