גיליון תשובות

מספר נבחן:_____

שאלה 1 (30 נקודות)

סעיף א (15 נקודות)

המתודה ()remove גם משנה את מצב האובייקט (command), וגם מחזירה ערך (query).
על פי עקרונות העיצוב שלמדנו בכיתה יש להפריד שאילתות מפקודות.
שאילתא המחזירה את גודל תוכן המאגר כבר קיימת (()getContent), ולכן נותר רק להגדיר את ()remove כפקודה – כ void.

```
//@INV: 0 <= getContent() <= getCapacity()

//@return the capacity of the pool
//@PRE: none
//@POST: none
long getCapacity();

//@return the content of the pool
//@PRE: none
//@POST: none
float getContent();

//@PRE: getContent() > 0
//@POST: getContent() == 0
void remove();

//@param addition > 0
//@PRE: getContent() + addition <= getCapacity()
//@POST: getContent() == @PRE(getContent()) + addition
void add(long addition);
```

סעיף ב (5 נקודות)

הגדרת בטיחות הינה שמירה על האינווריאנטה של האובייקטים במהלך ההרצה.
האובייקט היחיד החשוף בפני מספר אובייקטים אקטיביים הינו מטיפוס WaterPool
הגישה למצב הפנימי במחלקה זו נעשית תחת סינכרון מלא, ולכן המחלקה בטוחה

סעיף ג (10 נקודות)

בכיתה עסקנו בשני תרחישים של חבק:
- כניסה למצב blocked בעקבות <u>סנכרון</u> הדדי של שני אובייקטים.
- כניסה למצב blocked בעקבות <u>המתנה</u> הדדית על שני אובייקטים.

מהתבוננות בתרשים של המערכת ניתן להבחין בתלות ההדדית של שלושת הת'רדים בשני מיכלי המים.

כאשר מיכלים אלו מלאים, ה Producer ממתין על הוספת המים למיכל המטוהרים. במידה וה Consumer ממתין להוסיף מים למיכל המים המשומשים, וה Purifer ממתין להוספת מים למיכל המים המטוהרים נקבל חבק.
מצב בו שני המכלים מלאים, ניתן על ידי סדרת הפעולות הבאה:

- ה Producer ממלא את מיכל המים המטוהרים
- ה Consumer מרוקן את מיכל המים המטוהרים
- ה Consumer ממלא את מיכל המים המשומשים
- ה Purifier מרוקן את מיכל המים המשומשים
- ה Producer ממלא את מיכל המים המטוהרים
- ה Consumer מרוקן את מיכל המים המטוהרים
- ה Producer ממלא את מיכל המים המטוהרים
- ה Consumer ממלא את מיכל המים המשומשים
- ה Consumer מרוקן את מיכל המים המטוהרים
- ה Producer ממלא את מיכל המים המטוהרים

←

- ה Producer ממתין על מילוי מיכל המטוהרים
- ה Consumer ממתין על מילוי מיכל המשומשים
- ה Purifier ממתין על מילוי המים המטוהרים

על זיהוי התרחיש ניתנו 8 נקודות
על תרחישי חבק שאינם תואמים להרצת המערכת, כפי שהוצגה במבחן (getCapacity()==0, addition>getCapacity(), מיכלים בעלי קיבולת שונה), ירדו שתי נקודות.

כל שביב הצעה לפתרון החבק, נתן את מלא הנקודות (2).
מניעת החבק אפשרית בכל מיני דרכים, ובפרט, שינוי תנאי המילוי של ה Producer למקרה בו כמות המים הכללית במערכת הינה 0 [אמנם, לפי חדו"א א' זה לא יקרה לעולם (ראו לדוגמא הפרדוקס של זנון), אך במערכת ממוחשבת בה ה float הינו בן 32 ביטים, נגיע די מהר למספר קטן בו כל 32 הביטים הינם 0].

```
class WaterController {
  private float _total;
  public synchronized float get() { return _total; }
  public synchronized void add(float c) { _total+=c; notifyAll(); }
}

class ProducingTask implements Runnable {
  ...
  private WaterController _waterConroller;
  ...
  public void run() {
    while (true) {
      synchronized (_waterConroller) {
        while (_waterConroller.get() > 0)
          _waterConroller.wait();
        _purifiedPool.add(_purifiedPool.getCapacity());
        _waterConroller.add(_purifiedPool.getCapacity());
      }
    }
  }
```

```
}

class PurifyingTask implements Runnable {
  ...
  private WaterController _waterConroller;
  ...
  public void run() {
    while (true) {
      float orig = _usedPool.remove();
      float purified = _purifier.purify(orig);
      _purifiedPool.add(purified);
      _waterConroller.add(purified - orig);
    }
  }
}
```

שאלה 2 (30 נקודות)

סעיף א (2 נקודות)

Stack

0

סעיף ב (6 נקודות)

Heap

5016	2
5012	*vtable
5008	1
5004	*vtable
5000	1500

1500	1504
5012	2500

Stack

2500	2504
5004	0

סעיף ג (6 נקודות)

1500	1504
3000	2500

2500	2504
2000	0

2000	2004
*vtable	1

3000	3004	3008
*vtable	2	20

5008	3000
5004	2000
5000	1500

סעיף ד (6 נקודות)

מרכיב ראשון בפתרון הוא לזהות שיש צורך ב EventsStack()~ , רעיונות אחרים לא התקבלו
מרכיב שני הוא לממש EventsStack()~ כראוי. בפרט פתרון למופע הספציפי של הבעיה (למשל למחוק רק את top_*) אינו מספק.

```
EventsStack::~EventsStack() {
        while (_top != 0) {
                pop();
        }
}
```

סעיף ה (10 נקודות)

Heap

1500	1504
5012	2500

1000	1004
5012	2000

Stack

2500	2504
5004	0

2000	2004
5004	0

5020	1000
5016	2
5012	*vtable
5008	1
5004	*vtable
5000	1500

```
EventsStack& EventsStack::operator=(const EventsStack &other) {

	// check self assignment
	if ( this == &other ) return *this;
	// clear
	while (_top != 0) {
		pop();
	}
	// copy
	EventNode *cursor = other._top;
	if ( cursor == 0) {
		_top = 0;
		return *this;
	}
	EventNode *newNode = new EventNode;
	_top = newNode;
	while (cursor->next != 0) {
		newNode->data = cursor->data;
		newNode->next = new EventNode;
		cursor = cursor->next;
		newNode = newNode->next;
	}
	newNode->data = cursor->data;
	newNode->next = 0;
	// return
	return *this;
}
```

שאלה 3 (30 נקודות)

סעיף א (2 נקודות)

Background

The method getMessages() published by the StompOperator interface must return messages received by the Stomp client and between the time the StompOperator subscribed to a queue until the messages are retrieved. The difference between this method of obtaining messages and the one we know from the regular Stomp protocol is that the client "pulls" the messages from the StompOperator. In contrast, the regular Stomp server pushes messages to the client, one by one, without ever storing them.

Answer

If the StompOperator calls the StompServer to read messages, 2 bad things will happen:

It will be blocked until a new message arrives.

It will not receive messages that were sent before the call to getMessages()
(there will be missed message).

סעיף ב (10 נקודות)

Implement the method StompOperatorImp.getMessages() to avoid the problem:
We need to avoid the 2 problems:
Keep a queue of messages that are received from the server until getMessages() is called.
Make sure we read the messages in a different thread that does not block the RMI object.

```
public class StompOperatorImpl
  extends java.rmi.server.UnicastRemoteObject
  implements StompOperator
 {
  protected PrintWriter _writer;
  protected BufferedReader _reader;
  private List<String> _msgs;

  // Define a listener task to receive the messages from the Stomp server.
  private class Listener implements Runnable {
    private MessageTokenizer _tok;
    public Listener() {  _tok = new MessageTokenizer(_reader, '\0'); }
    public void run() throws IOException {
      while (_tok.isAlive())
        synchronize (_msgs) { _msgs.add(_tok.nextToken()); }
    }
  }

  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"));
    _msgs = new ArrayList<String>();
    // Construct listener
    new Thread(new Listener(_reader)).start();
  }

  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 {
     // Snapshot copy of the messages and reset it.
     List<String> res;
     synchronized (_msgs) {
        res = new ArrayList<String>(_msgs);
        _msgs.clear();  // reset the accumulator to receive new messages.
     }
     return res;
  }
}
```

A possible optimization was to start the listener thread only when we are asked to subscribe to a queue (as long as we did not subscribe we know we will not get messages from the stomp server).

Common Errors:
- Using non-blocking IO is not useful.
- Not using a message tokenizer (the input reader must tokenize the messages).
- Not synchronizing the access to the msgs list (between the listener and the getMessages call from the RMI thread).
- Not resetting the content of the msgs list after returning its content.

סעיף ג (3 נקודות)

```
Public class StompClientRMI {
Public class StompClientRMI {
   Public static void main(String[] args) {
      Try {
         // Get the address IP/port of the Stomp server from args
         // you could also have assumed they are "known" constants.
         // But they CANNOT be the same as the rmiregistry address.
         String stompHost = args[0];
         int port = Integer.parseInt(args[1]);
         StompConnector c = new StompConnectorImpl (stompHost, port);
         // Publish the connector to the rmiRegistry at the given address
         Naming.rebind("rmi://132.23.5.8:2010/StompConnector", c);
      } catch (Exception e) { e.printStackTrace(); }
   }
}
```

סעיף ד (6 נקודות)

```
public class StompClient {
```

```
public static void main(String[] args) {
  try {
    StompConnector c = (StompConnector)Naming.lookup("rmi://132.23.5.8:2010/StompConnector");
    StompOperator s = c.connect("user", "password");
    s.subscribe("q1");
    s.subscribe("q2");
    s.send("q3", "Suzy se");
    Thread.sleep(60000);
    for (String s : c.getMessages())
      System.out.println(s);
    } catch (Exception e) { e.printStackTrace(); }
  }
}
```

סעיף ה (6 נקודות)

The question defines exactly what is meant by "communication operation": sending a framed TCP message from one process to another.

TODO1:
StompClientConnector c = (StompClientConnector)Naming.lookup("rmi://132.23.5.8:2010/StompConnector");

Communication:
1. From StompClient to rmiRegistry (lookup) (send a query)
2. From rmiRegistry to StompClient (send the reply)

TODO2:
StompOperator s = c.connect("user", "password");

Communication:
1. From StompClient to StompClientRMI (send the "connect" message with its parameters serialized)
2. From StompClientRMI to StompServer (sends the "CONNECT" stomp frame)
   There is no reply from stompserver to StompClientRMI as explained in the question.
3. From StompClientRMI to StompClient: return value (reference to the StompOperator object).

Total: 2 + 3 = 5 TCP communication operations.

סעיף ו (3 נקודות)

Q: Does the number of TCP communication operations change if we change the implementation of the StompServer from Reactor to thread-per-connection (multiclient)?

A: The difference between the concurrency models (reactor/thread per connection) of the server is an internal implementation issue of the server. It does not affect in any way the implemented protocol. So there cannot be any difference in the TCP traffic.

שאלה 4 (10 נקודות)

סעיף א (5 נקודות)

```
// 3 tables are needed to create a normalized data model (no redundancy)
CREATE TABLE PlaneModels (
   Model varchar(20) PRIMARY KEY,
   TechnicalSpec varchar(10000)
   Capacity integer)

CREATE TABLE Planes (
    ID integer PRIMARY KEY,
    Model varchar(20) FOREIGN KEY REFERENCES PlaneModels(Model))

CREATE TABLE Flights (
   ID integer PRIMARY KEY,
   Destination varchar(100),
   Terminal varchar(20),
   ExitGate varchar(20),
   ExitDate date,
   ExitTime time,
   PlaneId integer FOREIGN KEY REFERENCES Planes(ID))
```

סעיף ב (5 נקודות)

```
Select PlaneModels.Capacity
From (Flights join (Planes join PlanesModels
                    on Planes.Model = PlanesModels.Model)
        on Flights.PlaneId = Planes.ID)
Where
   Flights.Destination = "Madrid" and Flights.ExitDate = '2010.02.01'
Order by PlaneModels.Capacity desc
```