2019 / 06 / 30
כ"ד באב תשע"ח

האוניברסיטה העברית בירושלים
ביה"ס להנדסה ומדעי המחשב ע"ש רחל וסלים בנין

מבוא לתכנות מונחה עצמים (67125)

(מועד א' בוקר)

מרצה: יואב קן-תור
מתרגלים: זיו בן-אהרון, נגה רוטמן, ג'ונה הריס
תעודת זהות של הסטודנט/ית: _____

הוראות:

- משך המבחן הינו **3 שעות**.
- המבחן הינו עם חומר סגור. אין להשתמש בכל חומר עזר או אמצעי חישוב שהוא, לרבות מחשבון.
- למבחן שני חלקים:
  - חלק א' – כולל 6 שאלות, מהן עליכם לענות על **5**. כל שאלה שווה 10 נקודות.
  - חלק ב' – כולל 2 שאלות קוד גדולות, ועליכם לענות על שתיהן. כל שאלה שווה 25 נקודות.

יש לענות על כל שאלות המבחן במחברת הבחינה.

שימו לב לנקודות הבאות:

- "Less is more" – רשמו תשובה קצרה ומדויקת, שלא תשאיר מקום לספק שאכן הבנת את החומר.
- ייתכן שהקוד המוגש יעבוד, אבל למבחן בתכנות מונחה עצמים זה לא מספיק. הקוד צריך להיות קריא ומתוכנן היטב. תעד את הקוד (אפשר בעברית) רק אם יש חשש שמישהו לא יבין אותו.
- זכור כי בעיצוב אין תשובה אחת נכונה, אבל בהחלט יש תשובות שהן לא נכונות. נקודות מלאות יתקבלו רק באם השתמשת באחת מהדרכים המתאימות ביותר.
- באם תענה על יותר שאלות מן הנדרש, רק הראשונות ייבדקו.
- אנו ממליצים לקרוא את כל המבחן מתחילתו עד סופו לפני תחילת הפתרון.
- המבחן כתוב בלשון זכר, אך מיועד לכלל סטודנטי הקורס במידה שווה.

בהצלחה!

חלק א'

- כולל 6 שאלות, מהן עליכם לענות על **5**. כל שאלה שווה 10 נקודות. הקפידו לנמק כאשר התבקשתם.
- חלק מן השאלות כוללות מספר סעיפים, אשר מופרדים לתיבות שונות במודל ומסומנים בהתאם (למשל, אם שאלה 1 כוללת שני סעיפים הם יסומנו כ"שאלה 1.1" ו"שאלה 1.2"). הקפידו לענות על כל חלקי השאלות.
- בשאלות הכוללות כתיבת קוד, הקפידו לכתוב קוד **יעיל ומובן**. רק מימושים כאלו יקבלו ניקוד מלא.
- בשאלות בהם התבקשתם למלא את ה-modifiers, ניתן למלא מילה אחת או יותר ממילה אחת. **תיבה ריקה תחשב כטעות - הקפידו למלא modifier בכל מקום שניתן!**

דוגמא ל-modifier בעל מספר מילים: `private static int x`

לעומת modifier בעל מילה אחת: `private int x`

שאלה 1

בחרו את האפשרות הנכונה עבור כל היגד בפסקה הבאה (כתבו את מספר המשפט ואת המילה הנכונה במחברת):

א. מחלקה פנימית (*inner class*) יכולה**לא יכולה** להגדיר (*declare*) מתודות סטטיות.
ב. מחלקה סטטית (*static class*) **יכולה**\לא יכולה להגדיר מתודות שאינן סטטיות.
ג. מחלקה פנימית **יכולה**\לא יכולה לגשת לשדות פרטיים (*private*) של המחלקה החיצונית לה.
ד. מחלקה פנימית **יכולה**\לא יכולה לגשת לשדות שהוגדרו *protected* במחלקה החיצונית לה.
הבהרה: אם למחלקה A קיימת מחלקה פנימית B אז נקרא ל A "המחלקה החיצונית של B"

שאלה 2

תארו בקצרה מהי משפחת התבניות **Factory** (הנקראת גם תבנית העיצוב **Factory**), לאיזה עיקרון של תכנות מונחה עצמים היא מתקשרת ותארו בקצרה שימוש אפשרי בה.

תשובה אפשרית:
משפחת התבניות **Factory** הינה משפחת תבניות אשר בעזרתן ניתן ליצר אובייקטים החולקים אב קדמון או ממשק.
משפחה זו מתקשרת לשני עקרונות שהכרנו:

- Open-Closed: מאפשרת לשמור על שיטות קודמות של יצירת אובייקטים תוך כדי הוספה של חדשות במידת הצורך
- Single Choice Principle: רשימת האפשרויות נשמרת במקום יחיד

נשתמש בה כאשר נרצה ליצור אובייקטים בצורה מורכבת, לדוגמא משחק החלליות בו קבענו איזה חללית ליצור באמצעות הקלט מהשחקן.

שאלה 3

1) מה תהיה תוצאת הקוד הבא:

```
public class Test
{
    public static void main(String[] args)
    {
        String obj1 = new String("oop");
        String obj2 = new String("oop");

        if(obj1.hashCode() == obj2.hashCode())
            System.out.println("hashCode of object1 is equal to object2");
        if(obj1 == obj2)
            System.out.println("memory address of object1 is the same as object2");
        if(obj1.equals(obj2))
            System.out.println("value of object1 is equal to object2");
    }
}
```

a. The code won't print anything

b. hashCode of object1 is equal to object2
memory address of object1 is the same as object2
value of object1 is equal to object2

c. memory address of object1 is the same as object2
value of object1 is equal to object2

d. hashCode of object1 is equal to object2
value of object1 is equal to object2

2) נמקו ב**קצרה** את תשובתכם לסעיף הקודם

תשובה אפשרית:

ההשוואה הראשונה היא של ההאש קוד, במקרה של מחרוזת פונקציית ההאש היא מספר שנובע באופן ישיר מערך המחרוזת (ולא מכתובת הזכרון, לו היה מדובר ברפרנס מסוג Object) ולכן מחרוזות עם אותו ערך חולקות את אותו מפתח האש. המקרה השני בודק השוואה של מיקום בזיכרון שהוא שונה מכיוון שאלו שני אובייקטים שונים. השלישי בודק שיוויון באמצעות המתודה equals ובמקרה של אובייקט מסוג סטרינג מבצעת השוואה של המחרוזות.

שאלה 4

1) השלימו את ה modifiers המינימלים (הכי מגבילים) ע"מ שהקוד הבא יתקמפל (מלאו את ה-modifier הנכון במחברת הבחינה על פי המספור המודגש באדום):

```
package pack1;
public class A
{
   1.protected int myInt;
   2.public A(int newInt){
       myInt = newInt;
   }

   3.protected int getInt() {
       return myInt;
   }
}

package pack2;
import pack1.A;

class B extends A
{
  4.default A myA;        // package private allows C() to access this
   B(){
       super(2);
       myA = new A(2);
       myInt = this.getInt() + 1;
   }
   A getMyA()
   {
       return myA;
   }
}
package pack2;
public class C extends B{
   5.private B myB;             // only the ctor uses this field.
   6.private C(){               // Only this class uses this ctor.
       myB = new B();
       this.myA = myB.getMyA();
   }

   public static void main(String[] args) {
       C c = new C();
   }
}
```

2) נמקו בקצרה את תשובתכם לסעיף הקודם
תשובה אפשרית:
הנימוק בהערות בגוף השאלה

שאלה 5

כתבו מתודה המקבלת מערך (*array*) של מספרים שלמים (*int*) ומחזירה את מספר המספרים השונים במערך. על חתימת הפונקציה להיות בפורמט הבא:

```
static int distinctElements(int[] arr);
```

לדוגמה:

- עבור המערך [1,5,2,1,1,5] הפונקציה תחזיר 3 (1, 5 ו-2 הם המספרים היחידים במערך)
- עבור המערך [0,1,3,7] הפונקציה תחזיר 4

הערות:

- נתון כי כל המספרים הנתונים נמצאים בטווח 0-1000000
- על הקוד שלכם להיות יעיל במקום ובזמן

Possible answer:

```
static int distinctElements(int[] arr) {
    HashSet<Integer> h = new HashSet<>();
    for (int i : arr) {
        h.add(i);
    }
    return h.size();
}
```

שאלה 6

נתונה המחלקה Point המייצגת נקודה במרחב דו-ממדי. המחלקה אינה ניתנת לשינוי.

```
public class Point {
   private int x, y;
   public Point(int x, int y) {
       this.x = x;
       this.y = y;
```

```
    }
    public double norm() {
        return Math.sqrt(x * x + y * y);
    }
    public Point add(Point other) {
        return new Point(this.x + other.x, this.y + other.y);
    }
}
```

כתבו מתודה המקבלת רשימה (*List*) של נקודות ומספר שלם **k** ומחזירה רשימה עם **k** הנקודות הקרובות ביותר לראשית הצירים מתוך הנקודות הנתונות. לפונקציה החתימה הבאה:

```
public static List<Point> getKClosestToOrigin(List<Point> points, int k)
```

(שימו לב – יש דוגמא בעמוד הבא)

norm	point
1.0	(1,0)
4.24	(-3,3)
1.41	(-1,1)
5.0	(5,0)
2.0	(0,2)

דוגמא:

אנו מייצגים נקודה בצורה (x,y). כאשר הפונקציה נקראת עם **k=3** ורשימת הנקודות:

```
{(1,0), (-3,3), (-1,1), (5,0), (0,2)}
```

ערך ההחזרה של הפונקציה יהיה רשימה באורך 3 עם הנקודות

```
{(1,0), (-1,1), (0,2)}
```

כי אלו הן 3 הנקודות בעלות הנורמה הקטנה ביותר (כפי שניתן לראות בטבלה)

הנחות והוראות נוספות:

- ניתן להניח שרשימת הנקודות הנתונה היא באורך לפחות k
- ניתן להניח כי אין שתי נקודות באותו המרחק מהראשית
- הנכם רשאים לכתוב מחלקות/מתודות עזר נוספות אם יש צורך בכך

Possible answer:

```
public static List<Point> getKClosestToOrigin1(List<Point> points, int k) {
    Comparator<Point> comparator = new Comparator<Point>() {
```

```
        @Override
        public int compare(Point p1, Point p2) {
            return Double.compare(p1.norm(), p2.norm());
        }
    };
    Collections.sort(points, comparator);
    LinkedList<Point> res = new LinkedList<>();
    for (int i = 0; i < k; i++) {
        res.add(points.get(i));
    }
    return res;
}
```

This solution uses anonymous class. The function can also be implemented using a concrete class implementing Comparator<Point> or by using a lambda expression.

חלק ב'

- קראו בעיון את שתי השאלות הבאות. קראו כל שאלה עד הסוף לפני תחילת הפתרון.
- לשאלות יכולות להיות מספר סעיפים ותתי-סעיפים.
- הקפידו לנמק את העיצוב שלכם עבור כל סעיף. תשובה ללא נימוק לא תקבל את מלוא הניקוד.
- כאשר אתם כותבים נימוק הקפידו לפרט אילו עקרונות ותבניות עיצוב (design patterns) היו בשימוש.
- הקפידו על עיצוב יעיל. העיצוב ייבחן על בסיס:
    1. שימוש בכלי המתאים ביותר מהכלים שנלמדו בקורס.
    2. שימוש במינימום ההכרחי של מחלקות וממשקים.
- בכל פעם שהוגדרה מתודה למימושכם, אם חתימתה חסרה, עליכם להשלים את ה-modifiers לפי שיקולכם.
- ניתן להשתמש בקוד Java של מחלקות שלמדנו עליהן (לדוגמא הפונקציה sort של Collections).

שאלה 1

הנביא יונה מאס בניהול מספר ארנקים שונים למעקב אחר המטבעות השונים שלו (ראה המשך). לשם כך החליט להטיל עליכם את המשימה לממש עבורו ארנק אחד שיבצע מעקב זה.

בעולם יש שתי קטגוריות של מטבעות:

1. מטבע פיאט (**FiatCurrency**)
2. מטבע וירטואלי (**CryptoCurrency**)

מספר דוגמאות למטבעות הם:

- מטבע פיאט בשם "שקל" שערכו בדולרים הוא 0.28
- מטבע פיאט בשם "דולר" שערכו בדולרים הוא 1
- מטבע וירטואלי בשם "ביטקוין" שערכו בדולרים הוא 10000

כמובן שיכולים להיות מטבעות נוספים.

הגדרות והסברים נוספים:

1. שימו לב כי כל מטבע משתייך לאחת מהקטגוריות ולאחת בלבד
2. לכל מטבע יש שם באנגלית
3. לכל מטבע יש ערך בדולרים
4. כל שני מטבעות עם שם זהה מייצגים את אותו המטבע. בפרט, אין שני מטבעות מקטגוריות שונות בעלי אותו שם
5. שם של מטבע חייב להיקבע בזמן יצירתו (בהמשך השאלה הסבר נוסף לגבי מי יוצר מטבעות)
6. ערך של מטבע יכול להיקבע בזמן יצירתו או לאחר מכן. אם הערך אינו נקבע בזמן היצירה יקבל המטבע ערך דיפולטיבי אפס
7. כל מטבע חשוף לאינפלציה (inflation)
    - הפעלת inflation על מטבע פיאט תגרום לירידת ערכו ל-90 אחוז מערכו המקורי (כלומר ערכו יוכפל ב-0.9)
    - הפעלת inflation על מטבע וירטואלי תגרום לירידת ערכו ל-80 אחוז מערכו המקורי (כלומר ערכו יוכפל ב-0.8)
8. כל ארנק יכול להכיל מספר מטבעות שונים (מספר מטבעות פיאט שונים ומספר מטבעות וירטואלים שונים). לכל מטבע, הארנק שומר את הכמות ממטבע זה הנמצאת בארנק

א) ממשו שתי מחלקות המייצגות את **FiatCurrency** ו- **CryptoCurrency**. באפשרותכם לממש טיפוסים (type) נוספים שיעזרו לכם במימוש המחלקות הנ"ל. זכרו כי כל מטבע תומך ב-4 פעולות:

הפעלת אינפלציה על המטבע (ראה הגדרות לתיאור התוצאה של פעולה זו)	`void inflation()`
מחזיר את שם המטבע	`String getName()`
מחזיר את ערך המטבע בדולרים	`double getUSDValue()`
משנה את ערך המטבע למספר הנתון	`void setUSDValue(double usdValue)`

הסבירו את החלטות המימוש שלכם בקצרה.

ב) בסעיף זה אתם נדרשים לממש את המחלקה **Wallet** אשר מייצגת ארנק. המחלקה צריכה לממש את כל המתודות הבאות:

בנאי היוצר ארנק חדש ריק	`Wallet()`
הוספה של *amount* מטבעות מסוג *coin* לארנק	`_____ addCoin( ____ coin, double amount)`
מחזיר את שווי כל המטבעות בארנק <u>בדולרים</u>	`double getWalletBalance()`

הסבירו את החלטות המימוש שלכם בקצרה.

<u>הבהרות:</u>

- שימו לב כי חלק מטיפוסי הפרמטרים וערכי ההחזרה חסרים ב API. עליכם להשלים אותם בצורה המתאימה ביותר לנתוני השאלה ובהתאם למימושכם
- ארנק מכיל <u>אוסף של מטבעות</u>, ו<u>כמות מכל מטבע</u> שנמצא בארנק
- על הארנק לתמוך בכל סוגי המטבעות
- הנכם רשאים להוסיף מתודות עזר נוספות שאינן מופיעות ב-API
- במתודה `addCoin`, אם המטבע הנתון כבר מופיע בארנק, הכמות הנתונה מתווספת על הכמות הקיימת שלו בארנק
- השווי בדולרים של מטבע *c* כלשהו שנמצא בארנק הוא הערך של המטבע בדולרים כפול הכמות ממטבע זה הנמצאת בארנק

ג) בשני הסעיפים, נמקו על עיצובכם בנוסף למימוש הנדרש.

ג.1) הרגולטור החליט שהבנק הוא האחראי הבלעדי על הנפקת (יצירת) מטבעות. כך יוכל הבנק לשלוט בערך המטבע ובתזמוני האינפלציה. כעת, כל מי שרוצה מופע של מטבע צריך לקבל אותו מהבנק. לשם כך הבנק הגדיר מתודה סטטית בשם `getCoin` (ראו חתימה מטה) אשר תחזיר מטבעות ע"פ דרישה. בנוסף, הבנק החליט שאם נוצר מופע של מטבע מסוים, אין טעם ליצור מופע נוסף שלו בעתיד, שכן מופע זה ייצג את אותו המטבע בדיוק.

ממשו את המתודה `getCoin` במחלקה **Bank** אשר מקבלת שם של מטבע רצוי ואת הטיפוס שלו ומחזירה מופע של המטבע המתאים. הבנק הגדיר enum שמייצג סוג מטבע.

```
public enum CoinType {FIAT, CRYPTO}
```

חתימת המתודה `getCoin` היא:

```
public static _____ getCoin(String name, CoinType type)
```

* שימו לב, גם כאן ערך ההחזרה חסר ועליכם להשלים אותו בהתאם למימושכם

<u>זכרו כי על פי החלטת הבנק, המתודה צריכה לממש את ההתנהגות הבאה:</u>

בהינתן שם של מטבע *name* וטיפוס *type*:

- אם בעבר נוצר מופע של מטבע עם שם *name*, יוחזר מופע זה
- אחרת, ניצור מופע של מטבע מטיפוס *type* ושם name עם ערך דיפולטיבי 0 ונחזיר אותו. בנוסף, נשמור מופע זה כדי שנוכל להחזירו בקריאות הבאות

עליכם לחשוב איך בדיוק לשמור מופע של מטבע שנוצר לשימוש בקריאות הבאות. הנכם רשאים להוסיף קוד נוסף במחלקה **Bank** מחוץ למתודה `getCoin`.

הקוד הבא מדגים את ההתנהגות הרצויה:

```
____ btc1 = Bank.getCoin("BTC", CoinType.CRYPTO);
____ btc2 = Bank.getCoin("BTC", CoinType.CRYPTO);
System.out.println(btc1 == btc2); // output is true
```

ג.2) כפי שהוזכר, הבנק אחראי גם על אינפלציית המטבעות. ממשו מתודה נוספת עבור המחלקה **Bank** בשם `inflation` שאחראית על הפעלת אינפלציה על כל אחד מהמטבעות הקיימים בעולם (כל המטבעות שנוצרו אי פעם על ידי הבנק).
חתימת המתודה:

```
static void inflation()
```

<u>הנחות והנחיות נוספות:</u>

- הפונקציה צריכה להפעיל את פעולת `inflation` על כל מטבע בהתאם לקטגוריה אליה משתייך המטבע (הגדרת פעולת אינפלציה לכל קטגוריה מופיעה בתחילת השאלה)
- פעולת אינפלציה אמורה להשפיע על כל המטבעות שנמצאים בארנקים (**Wallet**) של אנשים
- ניתן להניח כי יצירת מטבעות מתבצעת רק בבנק, במתודה `getCoin` שכתבתם בסעיף **ג.1**

Possible answer:
A) FiatCurrency and CryptoCurrency are both specific types of currency, that share similar data members and methods implementation. Therefore it is best to create another class - Currency - that declares their shared data members and implements their mutual methods. Both classes will inherit from that class. Since a currency that is not of type Fiat or Crypto has no meaning, Currency will be abstract. All methods in the API can be implemented in Currency except for inflation, that will be declared abstract and left for implementation in sub-classes. Currency also overrides equals and hashCode according to the currency's name (which is its identity) so we can insert it into a hash-based data structure in the next section (Wallet).

B) The wallet should hold a collection of currencies, each is associated with an amount. For that, we use a HashMap where each key is of type Currency (and therefore can be any of Fiat/Crypto) and the value is the amount of that currency in the wallet (Double). Calculating the wallet's balance is done by iterating all currencies in the wallet (all the keys in the map), multiplying each currency's USD value by its associated amount and sum it all up.

C) The Bank should store every created Currency, in case it is requested again in the future. For that we'll have a static data structure in the class in which we'll store all currencies (the DS must be static since getCoin is static). Since getCoin gets as an argument the name of the currency, which is its identity, we create a map where each key is a currency's name and the value is the Currency object (in practice, one of FiatCurrency/CryptoCurrency). If a given name already exists in the map as a key - we return its value. Otherwise we create a new Currency according to CoinType, store it in the map and return it. The inflation method is simple - just iterate all created currencies (all values in the map) and apply their inflation method. Since the bank holds any instance it ever created, and currencies are only created in the bank, this will affect every Currency in the program, specifically, all currencies in Wallets.

```
// ----- Section A -----


abstract class Currency {


    private String name;
    private double usdValue;


    Currency(String name, double usdValue) {
        this.name = name;
        this.usdValue = usdValue;
    }


    abstract void inflation();
```

```
    String getName() { return this.name; }

    /** get unit value in USD */
    double getUSDValue() { return this.usdValue; }

    /** set unit value in USD */
    void setUSDValue(double usdValue) { this.usdValue = usdValue; }

    @Override
    public boolean equals(Object o) {
        return o instanceof Currency && ((Currency) o).name.equals(this.name);
    }

    @Override
    public int hashCode() { return this.name.hashCode(); }
}


class FiatCurrency extends Currency {
    FiatCurrency(String name, double value) {
        super(name, value);
    }

    FiatCurrency(String name) {
        this(name, 0);
    }

    @Override
    void inflation() { this.setUSDValue(this.getUSDValue() * 0.9); }
}
```

```
class CryptoCurrency extends Currency {
    CryptoCurrency(String name, double value) {
        super(name, value);
    }
    CryptoCurrency(String name) {
        this(name, 0);
    }

    @Override
    void inflation() { this.setUSDValue(this.getUSDValue() * 0.8); }
}

// ----- Section B -----

class Wallet {
    private HashMap<Currency, Double> walletCoins;
     Wallet() {
        this.walletCoins = new HashMap<>();
    }

     void addCoin(Currency c, double amount) {
        if (!this.walletCoins.containsKey(c))
            this.walletCoins.put(c, amount);
        else {
            double currentValue = this.walletCoins.get(c);
            this.walletCoins.put(c, currentValue + amount);
        }
    }

     double getWalletBalance() {
```

```
        double sum = 0;
        Set<Currency> allCurrencies = this.walletCoins.keySet();
        for (Currency c : allCurrencies) {
            double coinAmount = this.walletCoins.get(c);
            double coinBalance = c.getUSDValue() * coinAmount;
            sum += coinBalance;
        }
        return sum;
    }
}

// ----- Section C -----

enum CoinType {FIAT, CRYPTO}



class Bank {

    private static HashMap<String, Currency> coins = new HashMap<>();

    public static Currency getCoin(String name, CoinType type) {
        if (coins.containsKey(name)) {
            return coins.get(name);
        }
        Currency c;
        switch (type) {
            case FIAT:
                c = new FiatCurrency(name);
                coins.put(name, c);
                return c;
            case CRYPTO:
```

```
                c = new CryptoCurrency(name);
                coins.put(name, c);
                return c;
            default:
                return null;
        }
    }

    static void inflation() {
        for (Currency currency : coins.values()) {
            currency.inflation();
        }
    }
}
```

שאלה 2

חברת מכשירי תקשורת מייצרת מכשירים למגוון לקוחות. המכשירים מתקשרים ביניהם **רק** באמצעות שרת (Server) המנתב את ההודעות למכשירים המתאימים. נתון כי לאובייקט **CommServer** יש את ה-API הבא:

מתודה היוצרת חיבור בין שני מכשירי קשר	`public boolean addConnection(CommDevice d1, CommDevice d2)`
מתודה המנתקת חיבור בין שני מכשירי קשר. אם הקשר לא קיים, לא קורה כלום	`public void removeConnection(CommDevice d1, CommDevice d2)`
מתודה ששולחת את ההודעה msg מהמכשיר d למכשיר אליו הוא מחובר. אם ההודעה נשלחה מוחזר true ואחרת מוחזר false	`public boolean sendMessage(CommDevice d, String msg)`

מהנדסי החברה כבר ממשו את מחלקת **CommServer** (עם ה-API הנתון) אבל צריכים עזרה במימוש מכשירי הקשר.

החליטו בחברה שמכשירי קשר יכולים לתקשר רק עם מכשיר קשר אחד בכל פעם, וכי אפשר ליצור קשר בין שני מכשירים רק כאשר שניהם לא מקושרים למכשיר אחר. לאחר יצירת קשר, המכשירים יכולים לשלוח איזה מספר הודעות שהם צריכים עד לניתוק אותו הקשר (לאחר מכן יהיה צריך לחדש את הקשר). כדי לאפשר ל Server לענות על הדרישות הללו הוחלט שכל מכשיר קשר חייב לעקוב אחר ה-API הבא:

מתודה המבקשת מה-Server להתחבר למכשיר קשר אחר. אם יצירת הקשר מצליחה, מוחזר true ואחרת false. **לא ניתן ליצור קשר בין מכשיר לעצמו!**	`public boolean connect(CommDevice d)`
מתודה המבקשת מה-Server לנתק את הקשר. אם המכשיר כבר לא מקושר, לא קורה כלום	`public void removeConnection()`
מתודה המאפשרת יצירת קשר חדש עם המכשיר d. המתודה נקראת על ידי ה-Server כשהמכשיר d מבקש להתחבר למכשיר, מחזירה true אם ניתן ליצור קשר ואחרת false	`public boolean acceptConnection(CommDevice d)`
מתודה המקבלת הודעה מה-Server. כאשר ה-Server מתבקש להעביר הודעה למכשיר, הוא קורא למתודה זו עם ההודעה שקיבל	`public void receiveMessage(String msg)`
מתודה שמבקשת מה-Server לשלוח הודעה למכשיר אליו מקושרים באותו הרגע. המתודה מחזירה true אם ההודעה נשלחה (כלומר, אם עדיין יש קשר עם מכשיר) ו false אחרת	`public boolean sendMessage(String msg)`
מתודה המדפיסה את ההודעה האחרונה שהתקבלה	`public void printMessage()`

מנהלי צוותי הפיתוח בחברה רוצים להיות מסוגלים לייצר מספר מכשירי קשר עם התכונות שפורטו בקלות.

דוגמא: נניח שיש שרת אחד *S* ומכשירי הקשר *A* ו-*B* רוצים לתקשר

```
>>    <A asks S to connect to B>
>>    <S asks if B accepts>
>>    <B isn't connected to anyone, so accepts the connection>
>>    <S connects between A and B>
>>    A: Hi B, it's me!
>>    <S sends message to B, returning true>
>>    B: Bye
<<    <S sends message to A, returning true>
>>    <B asks S to disconnect from A>
>>    <S disconnects A and B from each other>
>>    A: B?
>>    <S sees that A is not connected and returns false>
>>    A: :(
>>    <S sees that A is not connected and returns false>
>>    <A asks S to disconnect from B (so it will be free for other
connections)>
>>    <S does nothing>
```

שימו לב לכאורה, שני השלבים האחרונים מיותרים, אבל הוחלט שהשרת לא חייב לדווח שמכשיר ניתק קשר למכשיר השני

הערות:

- אין צורך לשמור היסטוריה של הודעות, רק את ההודעה האחרונה שהתקבלה
- שימו לב ש **CommServer** קורא ישירות ל-*receiveMessage* ול-*acceptConnection* בתזמונים הרלוונטיים
- שימו לב, השרת לא מדווח למכשיר קשר אם ניתקו אותו

א) ממשו את מכשיר הקשר הבסיסי ביותר על פי ה-API הנתון (מכשיר הקשר בעל התכונות הבסיסיות בלבד).

הסבירו בקצרה איך המימוש שלכם מאפשר יצירת מגוון מכשירי קשר עם התכונות הנדרשות.

ב) החברה קיבלה תלונות מלקוחותיה על כך שהיא לא מאפשר הצפנה בתקשורת בין המכשירים שלה. המהנדסים הראשיים של החברה הבחינו שמדובר בפגיעה בפרטיות הלקוחות שלה והחליטו להוסיף תוסף המאפשרת הצפנת הודעות הלקוחות על פי אלגוריתם הצפנה לבחירתם. הוחלט כי לתוסף החדש הבנאי הבא:

```
public EncoderDevice(CommDevice d, HashMap<Character, Character> encoder,
                     HashMap<Character, Character> decoder);
```

עזרו למהנדסים להוציא מוצר חדש המאפשר הוספת הצפנה למכשיר קשר שהלקוח כבר קנה (בלי לשלוח לו מחלקת מכשיר קשר חדשה!). על המכשיר החדש להיות מסוגל להצפין הודעות.

האם שיטת ההצפנה שהציעו מהנדסי החברה (בעזרת HashMap) עונה על עקרונות העיצוב שנלמדו בקורס? ספציפית, האם זוהי השיטה הכי כללית/גנרית? אם לא, הציעו עיצוב (design) מוצלח יותר. נמקו בקצרה.

הערות:

- כאשר המכשיר שולח הודעה, עליו להצפין אותה על ידי החלפת כל תו בהודעה בתו המתאים לו ב-encoder (הנתון באתחול)
- כאשר המכשיר מקבל הודעה, עליו לפענח כל הודעה באותו אופן על ידי decoder
- הניחו שכל לקוח יודע שלכל תו צריך להיות תו מקביל גם ב-encoder וגם ב-decoder

ג) החברה החליטה שמעבר לתוסף ההצפנה, יש להוסיף את האופציה לאימות הקשר בין מכשירים בעת יצירת החיבור ביניהם. הוחלט לממש תוסף נוסף שיתאים לכלל מכשירי החברה, לפיו כאשר מכשיר א' יוצר קשר עם מכשיר ב' על שני המכשירים לשלוח קוד אימות (בצורת הודעה) למכשיר השני. בקבלת ההודעה, כל מכשיר מוודא שהקוד שקיבל הוא אכן הקוד השמור ליצירת קשר עמו, ואם הקוד שגוי הקשר מיד מתנתק.

התוסף החדש צריך לממש את הבנאי הבא:

```
public AuthenticatorDevice(CommDevice d, String sendKey, String
                           receiveKey);
```

עזרו למהנדסים לממש את התוסף החדש.

הסבירו בקצרה כיצד ניתן להשתמש בשני התוספים במקביל וגם בנפרד. למשל, איך מתייחסים למקרה בו לקוח A משתמש רק במנגנון האימות, לעומת לקוח B שהחליט שהוא רוצה גם אימות וגם הצפנה. האם הצורה שבה מימשתם את הקוד הינה קלה להרחבה (extensible; כלומר קל ליצור תוספים נוספים)?

הערות:

- כפי שתואר, כשמכשיר A יוצר קשר עם מכשיר B, ההודעה הראשונה ש B מצפה לקבל היא sendKey וההודעה הראשונה שמכשיר A מצפה לקבל היא receiveKey

דוגמא לתהליך יצירת הקשר:

נניח שהקוד הוא שהמכשיר שיוצר את הקשר צריך לשלוח הוא ***itsme*** וקוד האימות שמקבל הקשר צריך לשלוח הוא ***iknow***. כאשר שני המכשירים מכירים באותם הקודים, תהליך יצירת הקשר יהיה

```
>>    <d1 connects to d2>
>>    d1: itsme                // d1 sends the correct sendKey
>>    <d2 verifies code>     // d2 checks that the sendKey is correct
>>    d2: iknow                // d2 sends the correct receiveKey
>>    <d1 verifies code>     // d1 checks that the receiveKey is correct
>>    d1: //some message     // regular communication can follow
```

אם d2 לא מכיר את הקוד הזה, נקבל

```
>>    <d1 connects to d2>
>>    d1: itsme                    // d1 sends the correct sendKey
>>    <d2 verifies code>          // d2 checks that the sendKey is correct
>>    <d2 disconnects from d1>  // d2 doesn't recognize the sendKey
```

לבסוף, אם d2 מתחזה

```
>>    <d1 connects to d2>
>>    d1: itsme                  // d1 sends the correct sendKey
>>    <d2 verifies code>  // d2 checks that the sendKey is correct
>>    d2: Hi!                    // d2 doesn't send the correct receiveKey
>>    <d1 verifies code>  // d1 checks that the receiveKey is correct
>>    <d1 disconnects from d2>  // d1 doesn't get the correct receiveKey
```

Possible Solution

Section A

This question should be implemented with Decorators. We have to use something like Decorators to make sure that we can add encoding and authentication, as well as other plugins, in the future. To this end, we will describe the following Interface:

```
public interface CommDevice {
   boolean connect(CommDevice d);
   void removeConnection();
   boolean acceptConnection(CommDevice d);
   void receiveMessage(String msg);
   boolean sendMessage(String msg);
   void printMessage();
}
```

Now it should be pretty simple to implement the most basic communication device:

```
public class StandardDevice implements CommDevice {
   String msg = "";
   CommServer curServer;
   CommDevice curDevice = null;

   public StandardDevice(CommServer server){ curServer = server; }

   public boolean connect(CommDevice d){
       if(curServer.addConnection(this, d)){
           curDevice = d;
           return true;
       } return false;
   }

   public boolean acceptConnection(CommDevice d) {
       if (curDevice == null) {
           curDevice = d;
           return true;
       } return false;
   }

   public void removeConnection(){
       curServer.removeConnection(this, curDevice);
       curDevice = null;
   }

   public void receiveMessage(String msg){ this.msg = msg;}

   public boolean sendMessage(String msg){ return
curServer.sendMessage(this, msg); }

   public void printMessage(){ System.out.println(msg); }
}
```

Section B

Once we understand that we need decorators, in addition to the written interface, the EncoderDevice is pretty easy to implement (only receiveMessage and sendMessage need to be changed)

```
public class EncoderDevice implements CommDevice {
```

```
   private CommDevice device;
   private HashMap<Character, Character> encoder;
   private HashMap<Character, Character> decoder;

   public EncoderDevice(CommDevice d,
                        HashMap<Character, Character> encoder,
                        HashMap<Character, Character> decoder){
       device = d;
       this.encoder = encoder;
       this.decoder = decoder;
   }

   public boolean sendMessage(String msg) {
       return device.sendMessage(msg.chars().mapToObj(c ->
encoder.get(c).toString())
               .collect(Collectors.joining("")));
   }

   public void receiveMessage(String msg) {
        device.receiveMessage(msg.chars().mapToObj(c ->
decoder.get(c).toString())
               .collect(Collectors.joining("")));
   }

   public boolean connect(CommDevice d) { return device.connect(d); }

   public void removeConnection(){ device.removeConnection(); }

   public boolean acceptConnection(CommDevice d) { return
device.acceptConnection(d); }

   public void printMessage(){ device.printMessage(); }
}
```

The method that was requested of us isn't the best. It would have been better to give the customers and Interface with the name of, say, Encoder, that has two methods:

```
String encodeMessage(String msg);
String decodeMessage(String msg);
```

In this manner, the customer has free reigns on the implementation of the encoding and the same functionality is achieved.

Section C

The implementation here is the same as the one for the encoder device, i.e. a decorator

```
public class AuthenticatorDevice implements CommDevice {
   private CommDevice device;
   private String sendKey, receiveKey;
   private boolean auth1 = false, auth2 = false;

   public AuthenticatorDevice(CommDevice d, String sendKey, String
receiveKey){
       device = d;
       this.sendKey = sendKey;
       this.receiveKey = receiveKey;
   }

   public boolean sendMessage(String msg) { return
device.sendMessage(msg); }

   public void receiveMessage(String msg) {
       if (!auth1 && msg.equals(sendKey)) {
           auth1 = true;
           auth2 = true;
           device.sendMessage(receiveKey);
       } else if(auth1 && !auth2 && msg.equals(receiveKey)) {
           auth2 = true;
       } else if (auth1 && auth2){
           device.receiveMessage(msg);
       } else {
           device.removeConnection();
       }
   }

   public boolean connect(CommDevice d) {
       auth1 = true;
       auth2 = false;
       if (device.connect(d) ) {
           device.sendMessage(sendKey);
           return true;
       }
       auth1 = auth2 = false;
```

```
        return false;
    }

    public void removeConnection(){
        device.removeConnection();
        auth1 = auth2 = false;
    }

    public boolean acceptConnection(CommDevice d) {
        if (device.acceptConnection(d)) {
            auth1 = false;
            auth2 = true;
            return true;
        } return false;
    }

    public void printMessage(){ device.printMessage(); }
}
```

Now we can use decorator over decorator to achieve an Authenticator/Encoder, and if we create new plugins we can just use them to decorate those before them.

אובייקטים נפוצים:

Class HashMap<K,V>	
V get(Object key)	Returns the value to which the specified key is mapped, or null if this map contains no mapping for the key
Set<K> keySet()	Returns a Set view of the keys contained in this map
V put(K key, V value)	Associates the specified value with the specified key in this map. If the map previously contained a mapping for the key, the old value is replaced.
V remove(Object key)	Removes the mapping for the specified key from this map if present
Collection<V> values()	Returns a Collection of the values contained in this map

Class HashSet<E>	
boolean add(E e)	Adds the specified element to this set if it is not already present
boolean contains(Object o)	Returns true if this set contains the specified element
boolean remove(Object o)	Removes the specified element from this set if it is present
Iterator<E> iterator()	Returns an iterator over the elements in this set

Class LinkedList<E>	
boolean add(E e)	Appends the specified element to the end of this list
void clear()	Removes all of the elements from this list
E poll()	Retrieves and removes the head (first element) of this list
E get(int index)	Returns the element at the specified position in this list

הרשימה נועדה לעזור ואינה מחייבת. ייתכן שקיימות פונקציות או סוגי אובייקטים שלא נמצאים ברשימה אבל כן צריך להשתמש בהם על מנת לפתור את המבחן