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' צהריים)

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.

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. ניתן להניח שהקלט הוא לא null.
- 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:

א. ממשק (*interface*) **יכול\לא יכול** להכריז (*declare*) על מתודות protected.

ב. מחלקה אבסטרקטית (*abstract class*) **יכולה\לא יכולה** להכריז על מתודות protected.

ג. ממשק **יכול\לא יכול** להגדיר שדות שאינם final .

ד. מחלקה אבסטרקטית **יכול\לא יכול** להגדיר שדות שאינם final.

ה. מתודה סטטית (*static method*) **יכולה\לא יכולה** להיות אבסטרקטית (*abstract*).

שאלה 2

הסבירו בקצרה את ההבדל בין **Comparator** לבין **Comparable**, כיצד ניתן להשתמש בכל אחד על מנת למיין מערך (*array*) ותארו יתרון וחסרון של כל שיטה.

Answer:

Both interfaces allow to apply order of some sort between instances of the same object by defining a relation between two objects. **Comparable** is used to define the natural order between two objects, and as such is implemented inside the class. Advantage – has access to private members, and supports the encapsulation principle. Disadvantage – cannot be changed during runtime. As the class implements its *compareTo(T object)* method, it is used by simply invoking the sort method on a group of objects of that type. **Comparator**, however, allows the user to define multiple different types of orders between objects. It can be implemented, e.g. as a standalone class or a lambda expression, which is passed as a parameter to the *Collection.sort* method (along with the data structure itself). Advantage – allows for the application of

multiple types of orders between the same objects (like with the Hotels you saw in exercise 3), which can be determined in runtime. Disadvantage – cannot access private members of the objects it compares.

שאלה 3

א) מהי תוצאת הקוד הבא:

```
public class Complex {
    private double r, i;
    Complex(double r, double i) {
        this.r = r;
        this.i = i;
    }
    Complex add(Complex other) {
        return new Complex(this.r + other.r, this.i + other.i);
    }
    public String toString() {
        return this.r + " + " + this.i + "i";
    }
    public static void main(String[] args) {
        Complex c1 = new Complex();
        Complex c2 = c1;
        c1 = new Complex(2, 5);
        Complex c3 = c1.add(c2);
        System.out.println(c3);
    }
}
```

1. $5.0i + 2.0$
2. $10.0i + 4.0$
3. שגיאת קומפילציה
4. שגיאת זמן ריצה

ב) נמקו בקצרה את תשובתכם לסעיף הקודם.

Once we define a constructor in Complex, the default constructor is not created automatically. Therefore, there is no constructor in Complex that accepts zero arguments, and the first line in the main function will result in a compilation error.

שאלה 4

א) השלימו את ה modifiers בקוד הבא על מנת שיודפס:

```
42
58
```

```
class OuterClass {
    1.static int y = 2;
    2.public firstNested nested = new firstNested();
    static class firstNested {
        3.public int a = 4;
        secondNested inner = new secondNested();

        4.public  class secondNested {
            void add(int num) {
                a += num;
                y += a + num;
            }
            void print_all() {
                System.out.print(a);
                System.out.print(y);
                System.out.println();
            }
        }
    }
    public static void main(String[] args) {
        OuterClass outer = new OuterClass();
        outer.nested.inner.print_all();
        outer.nested.inner.add(1);
        outer.nested.inner.print_all();
    }
}
```

ב) נמקו בקצרה את תשובתכם לסעיף הקודם.

Possible explanation:

y must be static so that ***firstNested*** (which is also static) will be able to access it. In addition, if ***a*** isn't static then ***nested*** can't be static, however if both of them are static then the code will compile and the correct numbers will be printed. Finally, if ***inner*** is static, then both ***a*** AND ***secondNested*** must also be static, otherwise the code will not compile.

שאלה 5

ממשו מתודה שמסמלצת טורניר בין זוגות של מתמודדים כאשר כל מנצח עובר שלב עד שנשאר אחד.

קלט: מערך (*array*) של אובייקטים מסוג `Challenger`.

פלט: המנצח של הטורניר (אובייקט מסוג `Challenger`).

האובייקטים המדמים את המתחרים (`Challenger`) **מומשו** (לא צריך לממש) והדבר היחיד שידוע עליהם זה שיש להם את המתודה:

```
public boolean winsAgainst(Challenger other);
```

המחזירה "true" רק אם האובייקט מנצח את האחר.

על חתימת המתודה שתממשו להיות:

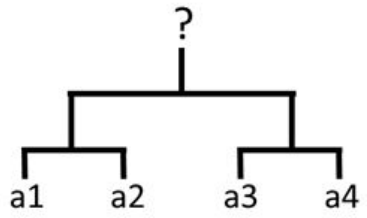
```
public static Challenger tournament(Challenger[] challengers);
```

דוגמא:

עבור שני מתמודדים `a` ו-`b` נסמן `a > b` אם `a` מנצח את `b`. אזי, עבור הקלט

```
Challenger[] challengers = {a1, a2, a3, a4};
```

כאשר `a4 > a1 > a2 > a3`, הטורניר שצריך להתקיים הוא (על פי סדר המשתתפים במערך):



שלב 1: `a1` נגד `a2` --> המנצח הוא `a1` וגם `a3` נגד `a4` --> המנצח הוא `a4`

שלב 2: `a1` נגד `a4` --> המנצח הוא `a4`

תוצאה: `a4` ניצח בטורניר ולכן הוא יוחזר

*(**שימו לב** - יכול להיות מצב בו הסדר שבו המתחרים מתמודדים אחד מול השני משנה את התוצאה. כלומר, יכול להיות מצב בו a>b>c וגם c>a)*

הנחות מקלות:

- הניחו כי מספר המתחרים הוא חזקה של 2
- הניחו כי לא יכול להיות תיקו בין שני מתחרים

Possible solution:

```
static Challenger tournament(Challenger[] classList){
   LinkedList<Challenger> tournamentList = new
LinkedList<Challenger>(Arrays.asList(classList));
   Challenger a1, a2;
   while (tournamentList.size() > 1){
       System.out.println(tournamentList);
       a1 = tournamentList.pop();
       a2 = tournamentList.pop();
       tournamentList.addLast(a1.winsAgainst(a2) ? a1 : a2);
   }
   return tournamentList.getFirst();
}
```

שאלה 6

עבור מערך (*array*) מספרים, נרצה למצוא את התת-מערך הארוך ביותר של מספרים עוקבים עם רווח זהה וחיובי ביניהם. כתבו מתודה המקבלת מערך של מספרים ומדפיסה את אורך התת-מערך הגדול ביותר ואת הרווח בין המספרים באותו תת-מערך בפורמט. לדוגמא, אם קיים תת-מערך כנ"ל באורך **7** עם רווחים של **2** בין כל המספרים בתת מערך, יודפס:

```
7,2
```

אם לא קיימת תת-מערך כמתואר, על המתודה להדפיס "0,0". החתימה של המתודה שתממשו חייבת להיות:

```
public static void biggestAscend(int[] nums);
```

דוגמאות:

- קלט: [0,1,2] פלט: 3,1 (התת-מחרוזת בדוגמא היא בעצם כל המחרוזת ולכן היא באורך 3, והרווח בין הספרות הוא 1)
- קלט: [10,0,1,2,4,6,8] פלט: 4,2
- קלט: [10,9,8,7,0] פלט: 0,0

Possible (not elegant) solution:

```
static void biggestAscend(int[] nums) {
   int diff = 0, prevDiff = 0, nDiff = 0;
   int[] diffVal = new int[100];
   for (int i = 1; i < nums.length; i++) {
       diff = nums[i] - nums[i-1];
       if (diff >= 0) {
           if (diff == prevDiff)
               nDiff++;
           else {
               if (nDiff > diffVal[diff])
                   diffVal[prevDiff] = nDiff;
               prevDiff = diff;
               nDiff = 1;
           }
       }
   }
   if (diff >= 0 && nDiff > diffVal[diff]) {
       diffVal[diff] = nDiff;
   }
   int retDiff =0, retArg = 0;
   for (int i = 0; i < diffVal.length; i++) {
       if (diffVal[i] > retDiff){
           retDiff = diffVal[i] + 1;
           retArg = i;
       }
   }
   System.out.println(retArg + "," + retDiff);
}
```

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

בשאלה זו נבנה מערכת פשוטה למידול חנויות. חנות מורכבת, בין היתר, ממוצרים (**Product**) וממוכר (**Seller**). כפי שידוע, מחיר מוצר מסוים יכול להשתנות בין חנות לחנות ואינו שווה בהכרח לערך המוצר בפועל. כך גם במערכת שלנו, לכל מוצר יש ערך גולמי וכל מוכר קובע מחיר למוצר כלשהו לפי פונקציית תמחור (שיכולה להיות תלויה למשל בערכו הגולמי של המוצר). למעשה, פונקציית תמחור היא פונקציה ממוצרים (**Product**) למספרים ממשיים.

לצורך מימוש הסעיפים בשאלה, נתונים לכם שני ממשקים: **PricingPolicy** ו- **Valuable**:

1. **Valuable**: ממשק המייצג אובייקט בעל ערך כספי (כמובן שלא כל האובייקטים בעולם הם בעלי ערך כספי, כמו למשל רעות)
2. **PricingPolicy**: ממשק המייצג שיטת תמחור לאובייקט בעל ערך (**Valuable**)

```
@FunctionalInterface
public interface Valuable {
    /**
     * @return the raw value of the object
     */
    double rawValue();
}
@FunctionalInterface
public interface PricingPolicy {
    /**
     * @return the price of the valuable v according to the policy
     */
    double price(Valuable c);
}
```

לא ניתן לשנות את הממשקים הנתונים!

בסעיפי השאלה אתם נדרשים לממש מחלקות שונות. אם לא צוינו במפורש חתימות של מחלקות/מתודות עליכם לקבוע אותם באופן המתאים ביותר להתנהגות הרצויה כפי שמוגדר בשאלה.

א) את המוצרים השונים בחנות נייצג בעזרת המחלקה **Product**. כל מוצר מאופיין על ידי שני פרמטרים:

- שם (מחרוזת)
- ערך גולמי (מספר ממשי)

על המחלקה **Product** להכיל בנאי המקבל שני ערכים אלו ובונה מוצר בהתאם:

```
Product(String name, double value)
```

<u>שימו לב:</u> ה-API של המחלקה **Product** ניתן לכם באופן חלקי. עליכם לחשוב מהי החתימה המלאה של המחלקה, האם נדרשות עוד מתודות ב- API ולממש אותן.

את המוכר נייצג בעזרת המחלקה **Seller**. לכל מוכר יש שיטת תמחור שונה (**PricingPolicy**) שנקבעת עבורו ברגע יצירתו. המחלקה **Seller** מממשת בנאי אחד ומתודה נוספת אחת:

`Seller(PricingPolicy p)`	בנאי היוצר מוכר עם שיטת תמחור *p*
`double priceOfProduct(Product prod)`	פונקציה המחזירה את המחיר של המוצר הנתון, ע"פ שיטת התמחור של המוכר

ממשו את שתי המחלקות **Product** ו- **Seller**. נמקו והסבירו את מימושכם.

ב) קים באג אין חזר לצפון קוריאה והחליט להקים שוק קפיטליסטי. כהתחלה הקים בסטה בשוק המקומי, והחליט לנקוט בשיטת תמחור שתשבור את השלטון - תמחור כל מוצר בשקל אחד פחות ממחירו הזול ביותר בשוק (כאשר כל המחירים נבחרו על ידי השלטון).

קים באג אין מבין בעגבניות והתקוממויות, אבל לא מבין בג'אווה. ממשו עבורו פונקציה סטטית המקבלת מערך של שיטות תמחור ומחזירה שיטת תמחור חדשה, הנותנת לכל מוצר את מחירו הנמוך ביותר מבין כל השיטות, פחות שקל אחד. חתימת המתודה הינה:

```
static PricingPolicy competitivePolicy(PricingPolicy[] policies)
```

נמקו בקצרה את מימושכם.

<u>הערות</u>:

- הינכם רשאים לממש מחלקות/מתודות עזר נוספות אם יש צורך בכך
- ניתן להניח כי המערך הנתון כארגומנט מכיל לפחות איבר אחד

ג) בסעיף זה נניח כי כבר מומשה עבורנו מחלקה המייצגת חנות:

```
class Store {
   /* ... data members ... */
   /* default constructor */
   Store() { /* initialization code */ }

   /* return the market cap of the store */
   double getStoreMarketCap() { /* return value code */ }
   /* ... more methods ... */
}
```

נוסף על מחלקה זו, נתונים לנו עוד ממשק ומחלקה ממומשת:

- **Business** ממשק המייצג עסק
- **StockExchange** מחלקה המייצגת בורסה, שמכילה אוסף של עסקים (**Business**)

```
public interface Business {
   /* return the market cap of the business */
   double getBusinessMarketCap();
}

class StockExchange {
   /* ... data members ... */
   /* default constructor */
   StockExchange() { /* initialization code */ }
   /* Add the given business to this stock exchange */
   void addBusiness(Business business) { /* add business code */ }
   /* ... more methods ... */
}
```

במימוש הנוכחי הנתון לנו, ניתן להכניס רק עסקים לבורסה (על ידי המתודה *addBusiness*). אנו נדרשים לבצע שינויים כך שגם חנויות (**Store**) יוכלו להיכנס בבורסה.

1. הסבירו מדוע לא ניתן להוסיף חנות (**Store**) לבורסה על ידי *addBusiness*

2. הציעו פתרון לבעיה שתיארתם בסעיף **ג.1** וממשו אותו, **כאשר אין באפשרותכם לשנות את המחלקה Store**

3. כתבו קוד קצר אשר יוצר בורסה וחנות ומוסיף את החנות לבורסה (תוך שימוש בפתרון שמימשתם)

Possible answer:

A) Product is a simple class with two data members that are given in its constructor. By looking at the Seller class, we understand that a seller should determine the price of a Product according to his PricingPolicy. However, PricingPolicy can only accept objects of type Valuable. Therefore Product must implement the Valuable interface, so a Seller can pass a Product to the price method of PricingPolicy.

B) The method competitivePolicy should return an object of type PricingPolicy, which is an interface. There are multiple ways of achieving that:

1. Create a class that implements PricingPolicy and return an instance of that class
2. Return an object of an anonymous class
3. Return a lambda expression

Either way, the object we return should give implementation for the price method of the PricingPolicy interface, that, given a Valuable v, finds the lowest price of v according to all the given policies, and return that price minus one.

C) The method addBusiness accepts only objects that implements the Business interface. So we'd like Store to implement Business, but we are not allowed to change it. The proper solution for that obstacle is to create another type that is-a Store (inherits from store) and implements Business, say StoreBusiness. This was we have an object that represents a store (it is a Store) and is also a Business, that can be passed to the addBusiness method.

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


public class Product implements Valuable {

    private String name;

    private double rawValue;


    Product(String name, double rawValue) {

        this.name = name;

        this.rawValue = rawValue;

    }
```

```
    public String getName() {
        return name;
    }

    @Override
    public double rawValue() {
        return this.rawValue;
    }
}

public class Seller {
    private PricingPolicy policy;

    Seller(PricingPolicy policy) {
        this.policy = policy;
    }

    double priceOfProduct(Product product) {
        return this.policy.price(product);
    }
}

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

class CompetitivePolicy implements PricingPolicy {
    private PricingPolicy[] policies;
    CompetitivePolicy(PricingPolicy[] policies) {
        this.policies = policies;
```

```
    }

    @Override
    public double price(Valuable v) {
        double min = policies[0].price(v);
        for (PricingPolicy policy : policies) {
            double price = policy.price(v);
            min = price < min ? price : min;
        }
        return min - 1;
    }

    static PricingPolicy competitivePolicy(PricingPolicy[] policies) {
        return new CompetitivePolicy(policies);
    }
}

// another possible and shorter solution with lambda expression:
class PricingPolicyUtils {
    static PricingPolicy competitivePolicy(PricingPolicy[] policies) {
        return v -> {
            double min = policies[0].price(v);
            for (PricingPolicy policy : policies) {
                double price = policy.price(v);
                min = price < min ? price : min;
            }
            return min - 1;
        };
    }
```

```
}


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

C.1) We can't add a *Store* to a *StockExchange* because the method *addBusiness* accepts only arguments of type *Business*, an interface that Store does not implement.

C.2) We can create a new class that inherits from *Store* and implements the *Business* interface. That way we have a type that represents a Store (with all of Store's fields and methods) and can be passed to the *addBusiness* method

```
class StoreBusiness extends Store implements Business {
    public double getBusinessMarketCap() {
        return super.getStoreMarketCap();
    }
}
public class Main {
    public static void main(String[] args) {
        StockExchange stockExchange = new StockExchange();
        StoreBusiness store = new StoreBusiness();
        stockExchange.addBusiness(store);
    }
}
```

שאלה 2

בספינת החלל USS Discovery יש מחסור חמור באנשי צוות לאחר היתקלות קשה עם משחתת קלינגונית. על הקפטנית נגה יוניקורן-בורנהאם לאייש במהירות את מחלקת ההנדסה על מנת לאפשר לדיסקאברי להמשיך במסעה ברשת המיציליאלית כדי להימלט מהאיום הקלינגוני המיידי.

מעבורת של צוערים (Cadets) נשלחה מהתחנה לבי עמוק 7 על מנת לסייע בתגבור הכוחות.

כל צוער הוא מופע של המחלקה הבאה:

```java
package USSCandidateSelection;
public class Cadet {
   private final String name;
   private int age;
   private int height;
   private List<Cadet> classMates = new ArrayList<>();
   private List<String> skills = new ArrayList<>();
   public Cadet(String name, int height, int age) {
       this.name   = name;
       this.height = height;
       this.age = age;
   }

   public void addClassMates(List<Cadet> classMates) {
      this.classMates.addAll(classMates);
   }
   public void addSkills(List<String> skills) {
       this.skills.addAll(skills);
   }
   public String getName() { return this.name; }
   public int getHeight() { return this.height; }
   public int getAge(){ return this.age; }
   public List<Cadet> getClassMates() {
       return new ArrayList<>(classMates);
   }
   public List<String> getSkills() { return new ArrayList<>(skills); }

}
```

א) קפטן יוניקורן-בורנהאם הטילה עליכם למצוא את המועמדים האופטימליים למחלקת ההנדסה. הדרישות מאנשי הנדסה:

- על איש הנדסה להיות מוכשר בתחומים הבאים (כלומר, על כולם להמצא ברשימת ה-*skills* שלו):

1. Warp Drives // = “WD”
2. Energy Pattern Rematerialization // = “EPR”
3. Dilithium Refining // = “DR”
4. Antimatter Reactors // = “AR”

(לנוחיותכם, השתמשו בקוד המקוצר הכתוב ליד כל אחד מהמקצועות)

- בנוסף, מאחר ובליבת מנוע העיוות חלק מהמכשירים נמצאים בגובה רם, על הצוערים להיות בגובה של לפחות 165 ס"מ

עליכם להוסיף לחבילה **USSCandidateSelection** טיפוס פומבי כלשהו בשם **CadetSelection** ובו שיטה עם החתימה הבאה:

```
public List<Cadet> getCadets(List<Cadet> candidates)
```

השיטה מקבלת את רשימת הצוערים ומחזירה רשימה ובה רק הצוערים שמתאימים למחלקת ההנדסה.

ענו ונמקו בקצרה (משפט או שניים) האם העיצוב שלכם מקיים או לא מקיים כל אחד מהעקרונות הבאים:

1. Composability
2. Decomposability
3. Open-Closed

שימו לב כי הפתרון שלכם לא מוכרח לקיים את כל העקרונות הנ"ל על מנת לקבל את מלוא הנקודות.

(שימו לב – המשך השאלה נמצא בעמוד הבא!)

ב) לנוחיותכם, הצוערים שנבחרו מועברים לספינה בטלפורטציה באמצעות פונקציית הקסם *teleportCadets*. על מנת להראות לקפטנית כיצד להשתמש בחבילה שלכם, השלימו את מחלקת ה-**MainClass** הבאה כדי שהקפטנית תוכל בקלות למצוא את אנשי ההנדסה המוצלחים ביותר לספינתה:

```
import java.util.*;

import USSCandidateSelection.*;


public class MainClass {

public static void main(String[] args) {

        List<Cadet> cadets = teleportCadets();

        //… complete function

}

private static List<Cadet> getMyFutureEngineers(List<Cadet> candidates) { ... }
```

- יש להשלים את תוכן הפונקציה הפרטית *getMyFutureEngineers* כך שתחזיר את חמשת המועמדים המתאימים ביותר
- לשם כך יש למיין את המועמדים מהמבוגר לצעיר (שכן הצוערים המבוגרים יותר בעלי יותר ניסיון)
- יש להשלים את תוכן פונקציית ה-main כך שבסיומה יודפסו הצוערים הנבחרים

הערות:

- אתם לא יכולים לשנות את חתימת השיטות, אבל רשאים להרחיב את ה-API של הטיפוסים כרצונכם, עם שמירה על עקרונות האנקפסולציה ו-minimal API
- אתם רשאים להגדיר בחבילה מחלקות וממשקים, פומביים או לא-פומביים, לפי שיקול דעתכם

(שימו לב – המשך השאלה נמצא בעמוד הבא!)

ג) מחלקת ההנדסה אוכלסה וספינת החלל דיסקאברי הצליחה להמלט מאזור העימות! כל הכבוד.

הקפטנית הייתה כל כך מרוצה מהפתרון שלכם שהיא רוצה כעת שתעזרו לה לאכלס גם את מחלקת האבטחה, על מנת לתגבר את הכוחות למקרה של עימות חדש. הדרישות מאנשי אבטחה:

- על איש אבטחה להיות מוכשר בתחומים הבאים:

1. Phaser Weapons // = PW
2. Bat'leth Dueling // = BD
3. Vulcan Death Grip // = VDG

- בנוסף, על מנת לאפשר ללוחמים לעבור בחללים הצרים שבספינה בעודם רודפים אחרי פולשים, לא ניתן לאפשר לצוערים מעל גובה 190 ס"מ להצטרף לשורות מחלקת האבטחה.

עדכנו את תכניתכם המקורית (פונקציית ה-main וכן כל טיפוס אחר שמימשתם אם יש בכך צורך), כך שתאפשר לאתר גם מועמדים למחלקת האבטחה. לשם כך:

- הוסיפו למחלקה **MainClass** את הפונקציה הסטטית:

```
private static List<Cadet> getMyFutureSecurity(List<Cadet>
                    candidates) { ... }
```

- ממשו אותה כך שתחזיר את 5 המועמדים הטובים ביותר. על מנת להיות איש ביטחון אפקטיבי, על המועמדים להיות גם בעלי יכולות חברתיות טובות. לשם כך - עליכם למיין את המועמדים לפי מספר החברים שיש לו במחזור שלו באקדמיה. המתודה *getClassMates* של כל צוער מחזירה מספר זה
- עדכנו את פונקציית ה-main כך שתדפיס בסופה גם את חמשת הצוערים המתאימים ביותר למחלקת האבטחה

ענו ונמקו בקצרה (משפט או שניים) האם העיצוב שלכם מקיים או לא מקיים כל אחד מהעקרונות הבאים:

1. Composability
2. Decomposability
3. Open-Closed

שימו לב כי הפתרון שלכם לא מוכרח לקיים את כל העקרונות הנ"ל על מנת לקבל את מלוא הנקודות.

בקובץ ה-זיפ המצורף (USS_Discovery_Solution.zip) תמצאו מספר גרסאות לפתרון. כל הגרסאות כתובות בגישת ה-streams and lambdas, אך כפי שאתם יודעים - כל פתרון כזה ניתן להמיר לפונקציות ומחלקות רגילות.

שלושת הפתרונות הם ברמת מורכבות הולכת וגדלה. כולם מתבססים על שני ממשקים - CadetSelection ו-CadetSort המהווים למעשה את אותם ממשקי Order/Filter אותם ראיתם בתרגיל 5. מעבר לממשקים, כל פתרון מציג רמת מורכבות הולכת וגדלה: -

1) מחלקות סלקטור וסורט נפרדות לכל מקרה (גיל, מספר חברים, מהנדסים ואבטחה)
2) מחלקת סורט נפרדת לכל מקרה (משול לשימוש במיון ללא מחלקה מבחינת מורכבות) ומחלקת סלקטור כללית המקבלת פרמטרים ומפלטרת לפיהם.
3) מחלקת סורט פרמטרית המקבלת קומפרטור (ממומש כ-method reference העטוף ב-reveresed, או כביטוי למבדה בו מיון בסדר הפוך מתבצע בקלות ע"י החזרת "מינוס" פרמטר ההשוואה) יחד עם מחלקת סלקטור כללית המקבלת פרמטרים ומפלטרת לפיהם.

הפתרון המועדף הוא כמובן הפתרון הכללי ביותר וניתן לממש פתרון דומה באמצעות מחלקות אבסטרקטיות המתוכננות היטב. פתרונות שאינם משתמשים בממשקים (או מחלקות אבסטרקטיות) או כאלו המציגים ירושה מיותרת יקבלו ניקוד חלקי.

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