2020 / 01 / 28
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')

מרצה: יואב קן-תור

מתרגלים: ג'ונה הריס, אלעד גרשון

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. החלוקה לסעיפים היינה לנוחות קריאה בלבד.

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.
- יש לתכנת בשפת JAVA
- אנו ממליצים לקרוא את כל המבחן מתחילתו עד סופו לפני תחילת הפתרון.
- המבחן כתוב בלשון זכר, אך מיועד לכלל הסטודנטיות והסטודנטים הלוקחים את הקורס במידה שווה.

בהצלחה!

חלק א'

- כולל 6 שאלות, מהן יילקחו חמש שאלות בלבד, בהן הניקוד הגבוה. 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 חלקי השאלה הוא לנוחות הפתרון בלבד.
- בשאלות הכוללות כתיבת קוד, הקפידו לכתוב קוד **קצר, פשוט ומובן ב-JAVA המשתמש בכלים המתאימים ביותר**. רק מימושים כאלו יקבלו ניקוד מלא.

שאלה 1

כדי ליצור אובייקט של מחלקה מקוננת (nested inner class) **[חייבים\לא חייבים]** ליצור אובייקט מהמחלקה העוטפת אותה (outer class)
מחלקה מקוננת סטטית (nested static class) **[יכולה\לא יכולה]** לגשת לכל המשתנים של המחלקה העוטפת אותה (פרטיים, מוגנים, סטטיים, לא סטטיים וכו')
מחלקה מקוננת **[יכולה\לא יכולה]** לגשת לכל המשתנים של המחלקה העוטפת אותה (פרטיים, מוגנים, סטטיים, לא סטטיים וכו')
מחלקה עוטפת **[יכולה\לא יכולה]** לגשת לכל המשתנים שהוגדרו בתוך מחלקה שמקוננת בה
מחלקה מקוננת **[יכולה\לא יכולה]** להיות מוגדרת כפרטית (private)

שאלה 2

תארו והסבירו את תבניות העיצוב Decoration ו- Delegation ותנו דוגמא לשימוש נכון באחת מהן (ניתן להסביר על סמך תרגיל שמימשתם בקורס).

פתרון:

בתבנית העיצוב Decoration אנו בונים מחלקה שהוא סוג של טיפוס קיים (ממשק/מחלקה אחרים), כאשר מחלקה זאת מבצעת את פעולותיה בעזרת אובייקט מאותו טיפוס קיים, הניתן לה לרוב בעת בנייתה. המחלקה החדשה היא מאותו סוג של האובייקט שאותו היא עוטפת, ומוסיפה לו התנהגות מסוימת (קישוט). לדוגמא, המחלקה ObjectOutputStream יורשת מהמחלקה OutputStream, ומקבלת בבנאי שלה אובייקט מסוג זה. ObjectOutputStream מרחיבה את ההתנהגות של האובייקט OutputStream שהיא מחזיקה (למשל, היא מאפשרת כתיבה אובייקטים, דבר שאינו מתאפשר ב OutputStream).
בתבנית העיצוב Delegation מחלקה מאצילה סמכויות למחלקה אחרת. גם כאן יש שימוש ב composition - מחלקה אחת מחזיקה אובייקט מטיפוס אחר. ההבדל ביניהן הוא שב-Delegation המחלקה המכילה אינה בהכרח חולקת טיפוס משותף עם האובייקט שהיא מחזיקה, אלא רק מעבירה אליו משימות. לדוגמה, כלי רכב יכול להאציל את הנעת הגלגלים למנוע, אך כלי רכב הוא אינו מנוע.

שאלה 3

הסבר ותאר את מנגנון הגנריות (Generics) ומדוע משתמשים בו באוספים (Collections).

פתרון:

מנגנון ה-Generics מאפשר לנו לכתוב מחלקות/מתודות עם טיפוס גנרים, אותם נחליף בטיפוסים קונקרטיים כאשר נרצה להשתמש במחלקות/מתודות אלו. ניתן להשתמש במחלקות גנריות במקומות שונים, ובכל פעם "לאתחל" את הטיפוס הגנרי עם טיפוס קונקרטי אחר. ניתן גם להוסיף הגבלות על טיפוס גנרי - למשל להגדיר טיפוס גנרי שיוכל להיות מאותחל

רק עם טיפוס היורש ממחלקה כלשהי, או מממש ממשק כלשהו.יכולת זו מאפשרת לנו לדרוש שהקוד יעבוד עם אותו סוג מבלי לקבוע ספציפית את הסוג.
באוספים, אנו רוצים לאפשר שימוש במבני נתונים שונים, עם טיפוסים שונים. בנוסף, אנו מעוניינים במבני נתונים שיכילו רק איברים מסוג מסוים. מנגנון ה Generics מאפשר לנו לעשות זאת - אנו יכולים ליצור מבני נתונים ולהגדיר בעת יצירתם את סוג האובייקטים שניתן להכניס אליהם. Java תוכל להתריע לנו למשל אם הכנסנו אובייקט מסוג כלשהו לאוסף שאינו תומך בסוג זה. יתרון נוסף ומהותי של מנגנון ה- Generics הוא type-safety: מנגנון זה מוצא שגיאות type כבר בזמן קומפילציה.

טעויות נפוצות:
כדי לתאר באופן מלא הייתם חייבים לציין:
1 - נגדיר מחלקה/מתודה עם טיפוס גנרי אותו נחליף לטיפוס קונקרטי כאשר נרצה להשתמש במחלקה/מתודה
2 - שניתן להגביל את הסוג הגנרי ולבצע עליו פעולות
3 - את מנגנון ה-type safety.
גם אם התכונה העיקרית שאנחנו מקבלים היא (3) עדיין צריך להבין מה נותן לנו את תכונה (3)
באוספים משתמשים בו בגלל:
1 - רק בעת יצירה אנו צריכים לקבוע מה הטיפוס שהאוסף יקבל.
2 - ואנחנו יכולים להגביל את האובייקטים שאנו נכניס ולקבל התרעה בזמן קימפול.

שאלה 4

הסבירו בקצרה מה המטרה של unit testing ותארו בקצרה את המשמעות של **3 מהאנוטציות** הבאות מ-JUnit:

1.Before	2.BeforeClass	3.Test	4.After	5.AfterClass	6.Test(expected =ExceptionA)

פתרון:
אם הקוד הוא encapsulated, לכל חלק בתוכנה יש תפקיד מוגדר היטב שהיינו רוצים לוודא שתמיד עובד, ללא תלות בחלקים אחרים של הקוד. המטרה של unit testing היא לעשות לבדוק את החלקים הללו באופן בלתי תלוי בחלקים השונים ולוודא שהתנהגות התוכנה היא הרצויה.
משמעות של Before: מתודה שרצה לפני כל אחת מהבדיקות
משמעות של BeforeClass: מתודה שרצה פעם אחת, לפני כל הבדיקות
משמעות של Test: מתודה שמריצה בדיקה
משמעות של After: מתודה שרצה אחרי כל אחת מהבדיקות
משמעות של AfterClass: מתודה שרצה פעם אחת, אחרי כל הבדיקות
משמעות של Test(expected=ExceptionA)a: מתודה שמריצה בדיקה ומצפה לקבל Exception מסוג ExceptionA
טעויות נפוצות:
מספר רב של תשובות לא התייחס ל-unit ב unit testing - כלומר, התשובה כללה משפט (או מספר משפטים) על כך ש-"צריכים לבדוק את הקוד" ואז את המשמעויות של האנוטציות. החלק הכי חשוב ב-unit testing היא שמחלקים את התוכנה למספר modules קטנים שאפשר לבדוק בנפרד ומוודאים שהמודולים האלה מתנהגים כרצוי. עוד טעויות נפוצות היו לבלבל בין Before ו-BeforeClass (או After ו-AfterClass) ולהגיד ש-Before/After רצים לפני/אחרי בדיקה ספציפית (הם רצים לפני/אחרי כל בדיקה).

שאלה 5

ממשו מתודה המקבלת מספר (int) ובודקת האם הוא מייצג מספר בינארי (מספר שכל ספרותיו הן 0 ו-1). על החתימה של המתודה להיות:

public static boolean isBinary(int num);

הערה: ניתן להניח שהמספר הנתון אינו שלילי ונמצא בטווח המתאים ל-int בג'אווה.

דוגמאות:

- הרצת המתודה על 1234512 תחזיר false
- הרצת המתודה על 110101 תחזיר true
- הרצת המתודה על 10000 תחזיר true
- הרצת המתודה על 100110002 תחזיר false
- הרצת המתודה על 0 תחזיר true

תשובה:

```
public static boolean isBinary(int num){
  Pattern p=Pattern.compile("[01]+");
  Matcher m=p.matcher(String.valueOf(num));
  return m.matches();
}
```

טעויות נפוצות

אפשר לפתור את השאלה עם רגקס כמו למעלה או לרוץ על הספרות על ידי חלוקה בעשר. הטעויות הנפוצות היו המרה למחרוזת מעבר תו תו ואז בדיקה של שוויון לספרות אפס או אחד או בדיקה באמצעות מבנה נתונים. שני הפתרונות הללו לא משתמשים בכלי המתאים ביותר (רגקס). הפתרון השני משתמש בכלי מאד מתקדם ומיותר בשביל הפתרון.
בנוסף טעות נפוצה הייתה לייצר תנאי מיותר בו בודקים אם m.matches מחזיר אמת אז החזר אמת אחרת החזר שקר.

שאלה 6

ממשו מתודה המקבלת מחרוזת (String) ובודקת האם סדר הסוגריים במחרוזת תקין. על החתימה של המתודה להיות:
public static boolean parOrder(String line);

הערה: ניתן להניח שאין תו יציאה (escape character) במחרוזת הנתונה (כלומר, אין דברים כמו '\n' במחרוזת).

הגדרה: סוגריים מורכבות מתו (character) פתיחה ותו סגירה. תווי הפתיחה האפשריים הם "(", "[" או "{" ותווי הסגירה המתאימים להם הם ")", "]" ו-"}", בהתאם. מחרוזת תקרא תקינה (מבחינת סוגריים) אם מתקיים:

1. אם קיים תו פתיחה, אז קיים תו סגירה מתאים (ולהפך) - כלומר, אם אין תו פתיחה, גם לא יהיה תו סגירה (ולהפך)
2. בין תו הפתיחה של סוגריים לתו הסגירה של אותם הסוגריים תופיע תת-מחרוזת אשר בעצמה הינה תקינה מבחינת סוגריים (שימו לב שזו יכולה להיות המחרוזת הריקה). כלומר, יכול להופיע שילוב של אותיות ומספרים (תווים מהסוג [a-zA-Z1-9] וכן תווים מיוחדים דוגמת #,%,&) וכל תת-מחרוזת שבעצמה תקינה מבחינת סוגריים

דוגמאות:

- הרצת המתודה על "no parentheses" תחזיר true
- הרצת המתודה על "good parentheses ()" תחזיר true
- הרצת המתודה על "({[]})" תחזיר true
- הרצת המתודה על "({[]})" תחזיר false
- הרצת המתודה על "(()" תחזיר false

תשובה:

```
public static boolean parOrder(String line) {
  if (line.length()==0) return true;
  LinkedList<Character> list = new LinkedList<>();
  char value, lastValue;
```

```java
for (int i = 0; i < line.length(); i++) {
    value = line.charAt(i);
    lastValue = list.isEmpty()?'N':list.peekLast();
    switch (value) {
        case '(':
        case '[':
        case '{':
            list.addLast(value);
            break;
        case ')':
            if (lastValue != '(') return false;
            list.removeLast();
            break;
        case ']':
            if (lastValue != '[') return false;
            list.removeLast();
            break;
        case '}':
            if (lastValue != '{') return false;
            list.removeLast();
            break;
    }
}
return list.isEmpty();}
```

טעויות נפוצות

מספר פתרונות לשאלה כללו ספירה של סוגי הסוגריים הפותחים ושל הסוגרים לראות האם המספרים האלה שווים. הבעייה בפתרון כזה שהוא יחזיר true גם עבור מחרוזות מהסוג "([)]", שהן לא תקינות ע"פ ההגדרה שניתנה. עוד פתרון שחזר על עצמו מספר פעמים היה להשתמש ב regex כדי לנסות לתפוס את הסוגריים החיצוניים ביותר, ואז ריקורסיבית לבדוק שמה שבין הסוגריים תקין (והשאלה הייתה כתובה באופן כזה באמת). הבעיות בפתרון הזה הן:

1. הרבה פעמים קשה לכתוב את ה-regex באופן המתאים ביותר, ואז מפספסים תווים
2. זמן הריצה של פתרון עם regex הוא גדול יותר מאשר הפתרון למעלה
3. המקום הדרוש לפתרון (memory complexity) גם הוא גדול יותר מאשר הפתרון למעלה

חלק ב'

- קראו בעיון את שתי השאלות הבאות. קראו כל שאלה עד הסוף לפני תחילת הפתרון.
- לשאלות יכולות להיות מספר סעיפים ותתי-סעיפים. החלוקה לסעיפים נועדה לנוחות הפתרון בלבד.
- שאלה ללא נימוק תאבד נקודות.
- כאשר אתם כותבים נימוק הקפידו לפרט באילו עקרונות ותבניות עיצוב (design patterns) בחרתם להשתמש ומדוע.
- הקפידו על עיצוב יעיל. העיצוב ייבחן על בסיס:
    - שימוש בכלי המתאים ביותר מהכלים שנלמדו בקורס.
    - שימוש במינימום ההכרחי של מחלקות וממשקים.
- בכל פעם שהוגדרה מתודה למימושכם, אם חתימתה חסרה, עליכם להשלים אותה לפי שיקולכם.
- ניתן להשתמש בקוד Java של מחלקות שלמדנו עליהן (לדוגמא הפונקציה sort של Collections).

שאלה 1

רקע

חברת התעופה באג-אין אירליינס החליטה להיעזר בכם לפיתוח מערכת ה-logging של מטוס הבאג-פורס-וואן החדש שלה. מערכת logging היא מערכת לתיעוד אירועים ומצבים שונים. התיעודים יכולים להתבצע בזמנים שונים במהלך הטיסה (מהמראה עד נחיתה). התיעוד נשמר בזמן אמת בכמה מקומות. מערכת זו מאפשרת לעקוב אחר מצב המטוס, לוודא האם המערכות תקינות, לגלות תקלות שונות ועוד.

כל מערכת logging כוללת את החלקים הבאים:

הודעות תיעוד	כל הודעת תיעוד log_message ברמה log_level צריכה להירשם ליעד (בהנחה והיעד תומך בסוג תיעוד זו) בפורמט הבא: **YYYY-MM-DD HH:mm log_level: log_message**
רמת תיעוד (log-level)	ישנם 4 סוגי תיעוד: 1. **DEBUG**. הודעות תחזוקה 2. **INFO**. מידע אודות אירועים כלליים בזמן הטיסה 3. **WARNING**. אזהרות לגבי מצב המערכות במטוס 4. **ERROR**. מידע על תקלות במערכות המטוס
יעד (Handler)	יעד אליו נכתבים תיעודים. לכל Handler יש ערך log-level מינימלי. Handler יכתוב תיעודים בעלי log-level גבוה (או שווה) לערך זה. יעדים אפשריים הם למשל "מסך הטייס" או "הקופסה השחורה"
לוגר (Logger)	האובייקט דרכו מבצעים תיעודים. אובייקט זה מכיל מספר יעדים שונים ומעביר הודעות לוג אליהם.

הבהרות:

- סדר חשיבות ה-log-level הינו ע"פ המספור לעיל, מהקטן (זניח) לגדול (חשוב ביותר)
- בשאלה זו נספק לכם API בו תוכלו להיעזר לצורך מימוש היעדים **"מסך הטייס" ו"הקופסה השחורה"**. עם זאת, זכרו כי לכל Logger יכול להיות **מספר יעדים גדול יותר**
- לכל יעד יכול להיות log level מינימלי **שונה**
- **YYYY-MM-DD HH:mm** הם התאריך והשעה של זמן התיעוד. השתמשו במחלקת העזר הנתונה TimeUtils כדי לקבל את התאריך והשעה בפורמט הרצוי (ראו API בהמשך)
- אובייקט Handler אחד יכול להיות במספר Loggerים

דוגמאות:

סדר סוגי התיעוד

להזכירכם סדר ה log-level הוא DEBUG < INFO < WARNING < ERROR:

- Handler שהוגדר עם log-level מינימלי WARNING יכתוב רק לוגים ברמות WARNING,ERROR
- Handler שהוגדר עם log-level מינימלי ERROR יכתוב רק לוגים ברמה ERROR
- Handler שהוגדר עם log-level מינימלי DEBUG יכתוב לוגים בכל הרמות

פלט אפשרי ממערכת logging של מטוס

כפי שניתן לראות, המערכת מספקת מידע שימושי לגבי ההתנהגות והמצב של המטוס בזמנים שונים.

2019-12-24 23:45 INFO: Preparing to take-off
2019-12-24 23:58 INFO: Starting take-off
2019-12-24 23:58 DEBUG: Engine temperature before take-off: 140
2019-12-25 00:14 INFO: Take-off complete
2019-12-25 00:28 WARNING: High engine temperature: 190
2019-12-25 01:09 ERROR: Passenger multimedia isn't working
2019-12-25 2:37 INFO: Preparing to land
2019-12-25 2:42 INFO: land complete

קוד עזר

באפשרותכם להשתמש במחלקות והטיפוסים הנתונים הבאים - אין צורך לממש אותם

public class TimeUtils	
public static String now()	מתודה המחזירה את הזמן הנוכחי בפורמט "YYYY-MM-DD HH:mm"

public class BlackBox	
public static BlackBox getBlackBox()	מתודה המחזירה את הקופסה השחורה של המטוס
public void write(String str)	מתודה הכותבת את המחרוזת הנתונה לקופסה השחורה

public enum LogLevel {DEBUG, INFO, WARNING, ERROR}

תזכורת: לכל Enum מוגדרת באופן אוטומטי המתודה compareTo המשווה את האובייקטים מסוג ה Enum ע"פ סדר הגדרתם

סעיף א'

נתון לכם ה- API של המחלקה Logger, תכננו וממשו את המחלקה, את הטיפוס Handler ואת היעדים "מסך הטייס" ו"קופסה שחורה".

public class Logger	
public Logger()	יצירת Logger חדש ללא Handlers
public void addHandler(Handler handler)	הוספת Handler חדש ל Logger זה
public void log(LogLevel level, String msg)	תיעוד ההודעה msg ברמה level. התיעוד מועבר לכל ה- Handlers ב- Logger זה

קטע הקוד הבא מדגים שימוש במערכת הרצויה וצריך לעבוד עם הקוד שמימשתם:

```
public static void main(String[] args) {
    Logger logger = new Logger();
    Handler blackBoxHandler = <create Handler with log-level DEBUG, writes to the black-box>
    Handler monitorHandler = <create Handler with log-level INFO, writes to the pilot's monitor>
    logger.addHandler(blackBoxHandler);
    logger.addHandler(monitorHandler);
    logger.log(LogLevel.DEBUG, "my debug msg");
```

```
    logger.log(LogLevel.INFO, "my info msg");
    logger.log(LogLevel.WARNING, "my warning msg");
    logger.log(LogLevel.ERROR, "my error msg");
}
```

Black box output (log-level=DEBUG)	Pilot monitor output (log-level=INFO)
2019-12-16 10:27 DEBUG: my debug msg 2019-12-16 10:27 INFO: my info msg 2019-12-16 10:27 WARNING: my warning msg 2019-12-16 10:27 ERROR: my error msg	2019-12-16 10:27 INFO: my info msg 2019-12-16 10:27 WARNING: my warning msg 2019-12-16 10:27 ERROR: my error msg

הבהרות:

- כתיבה למסך הטייס מתבצעת באמצעות מתודת ההדפסה של ג'אווה - System.out.print.
- כתיבה לקופסה השחורה מתבצעת באמצעות המחלקה **BlackBox** שתוארה לעיל.

סעיף ב'

בחברת התעופה הבינו כי ישנם Handlers אשר הכתיבה אליהם דחופה יותר מ- Handlers אחרים. לדוגמה, היינו רוצים שכתיבה למסך הטייס תתבצע לפני כתיבה לקופסה השחורה, ללא קשר לסדר הוספת ה- Handlers.
לשם כך החליטו להוסיף מנגנון ל- Logger אשר יאפשר לציין את העדיפות (priority) של כל Handler:

- העדיפות של Handler מיוצגת ע"י מספר שלם כלשהו (int).
- כאשר מבצעים פעולת log, ה- Logger צריך לכתוב לכל ה- Handlers שלו ע"פ העדיפויות שלהם בסדר יורד.

שנו את המתודה addHandler במחלקה Logger כך שתהיה עם החתימה הבאה:

public void addHandler(Handler handler, int priority)	הוספת Handler חדש ל Logger זה עם עדיפות **priority**

ממשו במחלקה Logger את השינויים הנדרשים כך שהמתודה log תתנהג בהתאם לדרישות, כלומר תכתוב ל- Handlers ע"פ הסדר הנדרש.

הבהרות:

- שימו לב - אין קשר בין log-level לבין priority. כלומר, log-level **לא** משפיע על ה-priority (ולהפך)
- לאחר השינוי בסעיף זה, קטע קוד ה-main מסעיף קודם יצטרך כמובן להשתנות כדי להתאים ל-API החדש של addHandler. התעלמו מכך
- ניתן להניח כי לא יוסיפו Handler יותר מפעם אחת ל- Logger כלשהו
- סדר יורד הינו מהגדול לקטן

סעיף ג'

הסבירו כיצד ניתן להרחיב את הקוד שכתבתם כך שיאפשר שימוש ב- Handlers חדשים נוספים (למשל Handler שכותב למגדל הפיקוח). תארו את הטיפוסים שנצטרך לשנות/להוסיף.
האם המימוש שלכם מקיים את עקרון ה- open-closed?
אין צורך לממש קוד בסעיף זה

פתרון:

```
public interface Writable {
  void write(String str);
}
```

```java
public class Monitor implements Writable {
  @Override
  public void write(String str) {
    System.out.print(str);
  }
}

public class BlackBoxWriter implements Writable {
  private BlackBox blackBox;

  public BlackBoxWriter() {
    this.blackBox = BlackBox.getBlackBox();
  }
  @Override
  public void write(String str) {
    this.blackBox.write(str);
  }
}

public class Handler {
  final private static String LOG_SEPARATOR = " ";
  final private static String NEWLINE = "\n";

  private Writable writable;
  private LogLevel level;

  public Handler(LogLevel level, Writable writable) {
    this.level = level;
    this.writable = writable;
  }

  private String formatMsg(LogLevel level, String msg) {
    return TimeUtils.now() + LOG_SEPARATOR +
        level.toString() + LOG_SEPARATOR +
        msg + NEWLINE;
  }

  public void log(LogLevel level, String msg) {
    if (level.compareTo(this.level) >= 0) {
      this.writable.write(this.formatMsg(level, msg));
    }
  }
}

public class Logger {
  // Objects added to a HashMap should override equals+hashCode. We ignore that in our solution
  Map<Handler, Integer> priorities;
  TreeSet<Handler> handlers;

  public Logger() {
    this.priorities = new HashMap<>();
    this.handlers = new TreeSet<>(new HandlerPriorityComparator());
  }
```

```java
    /**
     * Adds the given handler with the given priority. Logs are written to Handlers by their priority - Handlers with
     * higher priority are written to first
     */
    public void addHandler(Handler handler, int priority) {
        this.priorities.put(handler, priority);
        this.handlers.add(handler);
    }

    public void log(LogLevel level, String msg) {
        for (Handler h : this.handlers) {
            h.log(level, msg);
        }
    }

    private class HandlerPriorityComparator implements Comparator<Handler> {
        public int compare(Handler h1, Handler h2) {
            return -priorities.get(h1).compareTo(priorities.get(h2));
        }
    }
}
```

סעיף א: שני היעדים המדוברים בשאלה, וכן יעדים אחרים שאולי יתווספו בעתיד, חולקים תכונות ופעולות משותפות. לכולם יש log-level מינימלי, מקום שאליו הם כותבים תיעודים ופורמט אחיד שבו הם כותבים. למעשה כל Handler מבצע פעולות "כתיבה" למקום כלשהו, וה- Handlers נבדלים רק במקום שאליו הם כותבים. לכן בפתרון זה החלטנו להשתמש בתבנית העיצוב Strategy: נבנה מחלקה המייצגת Handler, וכל Handler יחזיק אובייקט שאליו אפשר לבצע כתיבה. את האובייקט הזה נייצג ע"י ממשק Writable, המייצג מקום שאליו אפשר לכתוב. ניצור שני אובייקטים המייצגים את מסך הטייס ואת הקופסה השחורה, כל אחד מהם יממש את הממשק Writable, ואת המתודה write שבתוכו, ע"פ היעד אותו הוא מייצג. כך, Handler אינו צריך לדעת להיכן או כיצד מתבצעת הכתיבה, והמימוש שלו אינו תלוי במקום שאליו כותבים.

סעיף ב: כעת אנו נדרשים לשמור על סדר כלשהו בין ה- Handlers הנמצאים בתוך Logger. נרצה אם כך להחזיק אותם במבנה נתונים ממוין, כאשר הסדר ייקבע לפי העדיפות שלהם. הנחה נוספת שניתנה לנו היא שאין כפילויות (לא מוסיפים Handler יותר מפעם אחת ל- Logger). לכן בחרנו במבנה הנתונים TreeSet. כדי להגדיר את הסדר ע"פ העדיפות של ה- Handlers, יצרנו Comparator שמשווה את ה Handlers ע"פ העדיפות שלהם. על מנת שה- Comparator ידע מהי העדיפות של כל Handler אנו חייבים לשמור גם את העדיפויות במבנה נתונים - זאת עשינו בעזרת מיפוי בין Handler ל- priority שלו. <u>שימו לב</u>: העדיפות אינה חלק מהתכונות של Handler, שכן כל Handler יכול להתווסף ל- Loggers שונים עם עדיפות שונה. לכן יש לשמור את העדיפויות באובייקט Logger.

סעיף ג: על מנת לתמוך בסוג חדש של Handler, כל שנצטרך לעשות הוא לממש אובייקט שמממש את הממשק Writable. לאחר מכן נוכל ליצור אובייקט Handler שאותו נאתחל עם האובייקט החדש שיצרנו.
המימוש שלנו אכן מקיים את עקרון ה open-closed: אין לנו כל בעיה להרחיב את הפתרון על מנת לתמוך בדרישות חדשות (למשל סוג Handler חדש). בנוסף, הקוד שלנו סגור לשינויים, שכן לא נדרשנו לבצע אף שינוי בקוד הקיים (אף מחלקה, ובפרט Handler, לא השתנתה בעקבות הדרישות החדשות)

<u>טעויות נפוצות:</u>

1. מיון אוסף מספר רב של פעמים: סטודנטים רבים מיינו את אוסף ה- Handlers בתוך אובייקט Logger בכל פעם שמתבצעת פעולת log או בכל פעם שמוסיפים Handler חדש. ביצוע מיון מספר פעמים הינה פעולה בזבזנית ובמקרה זה היה נדרש לבחור מבנה נתונים ממויין
2. מיון HashMap: פתרונות רבים ניסו להשתמש בקוד הממיין אובייקט HashMap. אובייקט זה בג'אווה הינו חסר סדר! לא ניתן להניח עליו סדר כלשהו או למיין אותו
3. שמירת מיפוי בין עדיפות (בתור מפתח) לבין Handler (בתור הערך) בתוך Logger. שמירת מיפוי כזה עלולה לגרום לנו לאבד Handlers אם נוסיף אחד חדש עם עדיפות זהה ל- Handler אחר שכבר קיים ב- Logger. בכל Map יכול להופיע מפתח כלשהו לכל היותר פעם אחת, ולהצביע על ערך אחד בלבד
4. שמירת עדיפות של Handler בתור שדה של Handler. החלטה זו הינה שגויה מבחינה עיצובית, שכן עדיפות של Handler היא בעלת משמעות רק בהינתן Logger כלשהו - זוהי העדיפות של Handler ב- Logger ספציפי כלשהו. בהוראות התרגיל כתוב במפורש כי "**אובייקט Handler אחד יכול להיות במספר Loggerים**". פתרון כזה לא מאפשר מצב בו Handler נמצא במספר Loggers שונים עם עדיפות שונה בכל אחד מהם
5. כפל קוד בין Handlers קונקרטים שונים. למשל בניית ההודעה בפורמט הנדרש, בדיקת log-level מינימלי, שדה log-level מינימלי, וכו'.

שאלה 2

<u>רקע</u>

מדינת ישראל רוצה לבנות מערכת תחבורה יעילה, אשר תורכב ממספר 'צורות' (Modes). אמצעי תחבורה אלו יוכלו לנוע בתנאי שטח שונים ואף באוויר. לכן, פנו ללא אחר מאשר פרופ' ריק. לאחר שזה דיבר עם שר התחבורה, הם הסכימו על רעיון חדש: רובוריקים אשר יוכלו לעבור בין מצבי תנועה. הרובוריקים יוכלו להסיע נוסעים ולשנות את צורתם על פי הצורך.

<u>שימו לב:</u> בשאלה זו נתעסק עם שתי צורות בלבד, אך העיצוב שתבחרו צריך לאפשר הוספה של צורות נוספות בקלות.

כידוע לכם, מערכת תחבורה הינה מלאת תקלות שונות. לכן, פרופ' ריק דורש שתבנו עץ ירושה של חריגות (**Exceptions**), ושתוסיפו חריגות במקומות המתאימים לכך.

<u>סעיף א'</u>

הטיפוס רובוריק (**RoboRick**) מוגדר כך:

public void changeMode(_____ newMode)	**מתודה המחליפה צורה אם יש מעל 50 אנרגריק.** מפחית 50 אנרגריק מהאנרגיה של הרובוריק ומשנה את צורתו לצורה החדשה newMode
public int getPassengers()	מתודה המחזירה את כמות הנוסעים ברובוריק
public void setPasngers(int passengers)	מתודה שמעדכנת את מספר הנוסעים (אין הגבלה על מספר הנוסעים)
public int getEnergRick()	מתודה המחזירה את מידת האנרגריק של הרובוריק

המחלקה מצב הנסיעה (**DriveMode**) מוגדרת כך:

public DriveMode()	בנאי המחלקה, מכניס 4 גלגלים
public void addWheels(int wheels)	מתודה המעדכנת את מספר הגלגלים ע"י ניסיון להוסיף כמות wheels לכמות הקיימת. **רובוריק לא יכול להכיל יותר מ-5 גלגלים**

public void removeWheels(int wheels)	מתודה המורידה כמות גלגלים נתונה. **ניתן להסיר רק מספר גלגלים קטן מהקיים.**

המחלקה מצב טיסה (**FlightMode**) היורשת מהמחלקה FlightModeBase מוגדר כך:

public FlightMode()	בנאי מחלקה
public void perpareForFlight()	מתודה המכינה את המטוס לטיסה. קיים לה מימוש גם במחלקה FlightModeBase. <u>הפעלתה יקרה מאד במשאבים.</u>
Public boolean isReadyForFlight()	מתודה הבודקת האם המטוס מוכן לטיסה.

ממשו את הטיפוסים **RoboRick, FlightMode, DriveMode** על כל פעולותיהם והוסיפו בנאים כנדרש. **השלימו את חתימות הפונקציה היכן שנדרש.**
להזכירכם, הוסיפו חריגות בכל מקום הדורש חריגה ושנו את חתימות הפונקציות בהתאם.

<u>הבהרות</u>

- הניחו כי מספר הנוסעים וכמות הגלגלים המוכנסת היא חיובית
- הרובוריק מתחיל במצב נסיעה (DriveMode) עם אנרגריק של 200
- המחלקה RoboRick אינה מאפשרת הוספת אנרגריק
- המחלקה FlightModeBase נתונה לכם ואין צורך לממשה. בין היתר, היא מכילה בנאי ברירת מחדל (default constructor) ואת המתודה perpareForFlight, ללא ארגומנטים

<u>סעיף ב'</u>

עכשיו הגיע הזמן לתזוזה של הרובוריקים! פרופ' ריק הבין שצריך to get swifty (לזוז) ולכן הוסיף עבורכם (ועבור מורטי) את הפעולות הבאות במחלקת **GetSwifty** (*אין צורך לממש*)

static void rickMotion(float x, float y, RoboRick rivka)	שיטה אשר מזיזה את הרובוריק לנקודת הציון הנתונה. **רובוריק מסוגל לנוע פחות מ-100 ק"מ בכל הפעלה.**
static float rickDistance(float x, float y, RoboRick rivka)	שיטה המחזירה מרחק בקילומטרים בין נקודת ציון למיקום הנוכחי של הרובוריק.

עליכם להשתמש בשיטות הללו ולממש את הפונקציונליות הבאה ב-RoboRick:

void move(float x, float y)	מתודה אשר מזיזה את הרובוריק לנקודת הציון הנתונה. **במצב נסיעה הרכב יזוז רק אם יש בו לפחות 4 גלגלים. במצב טיסה הרובוריק יזוז רק אם הוא הוכן לטיסה לפני כן.**

<u>שימו לב:</u> הרובוריקים עצמם לא מטפלים במיקומם. כל תזוזה ושינוי מקום של רובוריק מתנהל בתוך המחלקה GetSwifty, שמומשה עבורכם.
להזכירכם, הוסיפו חריגות בכל מקום הדורש חריגה ושנו את חתימות הפונקציות בהתאם.

<u>סעיף ג'</u>

אחרי שפרופ' ריק סיים לבנות את הרובוריק שלו, הוא תהה: "מה יותר טוב מרובוריק אחד?" וענה לעצמו: "הרבה רובוריקים!". לכן, הוא בנה רשימה מקושרת של רובוריקים. לצערו, הרשימה הייתה מבולגנת במצבים ובנוסעים. לכן רצה ריק למיינם לפי סדר מסוים.

הסדר הטבעי עליו חשב הוא שכל רובוריק תלוי במצב (mode) בו הוא נמצא. לשם כך נגדיר את מקדם המצב, ונקבע כי הוא שווה ל-3 אם הרובוריק במצב טיסה, ול-2 אם הרובוריק במצב נסיעה. כעת נגדיר את ערך הרובוריק, דרך מקדם המצב כפול מספר הנוסעים. לדוגמא:

- ערך רובוריק במצב טיסה עם שני נוסעים יהיה 3*2=6
- ערך רובוריק במצב נסיעה עם ארבעה נוסעים יהיה 2*4=8

לעומתו, שר התחבורה החדש ביצה-אל סמוטפור (שיודע שהוא לא יישאר להרבה זמן, ויבוא עוד שר בקרוב אחריו) החליט על סדר מיון משלו, שעובד רק על פי מצב הרובריק (תחילה המטוסים, אחריהם כל הרכבים, ולבסוף כל מה שנשאר)

הסבירו מתי תשתמשו ב-comprator ומתי ב-comparable למיון הרשימה ומדוע. השלימו את הפונקציות sortByValue הבאה הממיינת רובוריקים לפי הנוסחה של פרופ' ריק, ואת פונקצית sortByMode עבור שר התחבורה.

```
public static void sortByValue(LinkedList<RoboRick> roboRicks){
        //**your implementation**
}
public static void sortByMode(LinkedList<RoboRick> roboRicks){
        //**your implementation**
}
```

פתרון

```
/**
* reperents a base mode that each mode has to have
* such as getting a value, method move
* @author OOP staff
*/
interface BaseMode extends Comparable<BaseMode>{
  int getValue();
  /**
   * try to move a robo rick into destination (x,y) if unsuccessful throw expectation
   * @param roboToMove what robo rick to move
   * @param x first coordinate
   * @param y second coordinate
   * @throws ExceptionRoboRick a general exception that has some inheritance
   */
  void move(RoboRick roboToMove,float x,float y) throws ExceptionRoboRick;
  default int orderInList(){return 0;}
  default int compareTo(BaseMode o){return Integer.compare(orderInList(),o.orderInList());}

}

class RideMode implements BaseMode {
  /**
   * max wheels available to have in Ride mode, constant
   */
  private static final int MAX_WHEELS=5;
  private int wheels;
```

```java
    public RideMode(){
        wheels=4;
    }

    public int getValue() {  return 2;  }
    public int orderInList() {return 1;    }

    /**
     * try to add a given wheels number to already wheels.
     * throws exception if wanted to add more than what we can add or we started moving
     * @param wheelsNumber addition to the wheels
     * @throws ExceptionRoboRickNotWheels extends of ExceptionRoboRick
     */
    void addWheels(int wheelsNumber) throws ExceptionRoboRickNotWheels {
        if(wheelsNumber+wheels>MAX_WHEELS)
            throw new ExceptionRoboRickNotWheels();
        wheels += wheelsNumber;
    }

    public void removeWheels(int wheels) throws ExceptionRoboRickNotWheels {
        if(this.wheels - wheels < 0)
            throw new ExceptionRoboRickNotWheels();
        this.wheels -= wheels;
    }

    /**
     * override the baseMode method move and try to move roboRick to (x1,y1), throws exception if don't have enough wheels
     * @param roboToMove what robo rick to move
     * @param x1 first coordinate
     * @param y1 second coordinate
     * @throws ExceptionRoboRickMoving being thrown if not enough wheels (4) are in ride mode
     */
    public void move(RoboRick roboToMove,float x1,float y1) throws ExceptionRoboRickMoving {
        if(wheels < 4)
            throw new ExceptionRoboRickMoving();
    }

}

class FlightMode extends FlightModeBase implements BaseMode{
    private boolean isPrepared;

    public FlightMode() {isPrepared=false;}

    public int getValue() { return 3; }
    public int orderInList() {return 2;}

    /**
     * put the flag on ture, and called the prepareForFlight
     */
    public void perpareForFlight(){
        isPrepared=true;
        super.perpareForFlight();
    }
    public boolean isReadyForFlight(){return isPrepared;}
```

```java
  public void move(RoboRick roboToMove,float x1,float y1) throws ExceptionRoboRickMoving {
    if(!isPrepared)
      throw new ExceptionRoboRickMoving();
  }
}

/**
* represent the roboRick and implements Comparable
*/
class RoboRick implements Comparable<RoboRick>{
  private static final int THRESHOLD=50;
  private static final int MAX_DISTANCE=100;
  private static final int INIT_ENERGY=200;
  BaseMode currentMode;
  private int numberOfPassengers=0;
  private int energy;
  public RoboRick(){
        currentMode=new RideMode();
    energy=INIT_ENERGY;
  }

  /**
   * try to move this roboRick to (x,y), throws exception if distance is too high or mode throws
   * @param x first coordinate
   * @param y second coordinate
   * @throws ExceptionRoboRick any exception that throw by RoboRick Exceptions
   */
  public void move(int x,int y) throws ExceptionRoboRick {
    if(getSwifty.rickDistance(x,y,this)>=MAX_DISTANCE)
      throw new ExceptionRoboRickDistance();
    currentMode.move(this,x,y);
    getSwifty.rickMotion(x,y,this);
  }
  public float getEnergRick(){return energy;}
  public int getPassengers(){return numberOfPassengers;}
  public void setPassengers(int passengers){this.numberOfPassengers=passengers;}


  public void changeModes(BaseMode changeTo) throws ExceptionRoboRick {
    if(energy<=THRESHOLD)
      throw new ExceptionRoboRickNotEnoughEnergy();
    energy-=THRESHOLD;
    currentMode=changeTo;
  }

  /**
   * compare to function that implements compareable compare 2 items by Rick compares
   * @param B other roborick
   * @return 1 if one bigger than other 0 if equal. -1 else.
   */
  public int compareTo(RoboRick B){
    return Integer.compare(currentMode.getValue() * numberOfPassengers
, B.currentMode.getValue() * B.numberOfPassengers);
  }
}
```

```java
public class main{
  /**
   * sort the robricks by value of rick.
   */
  public static void sortByValue(LinkedList<RoboRick> roboRicks){
     Collections.sort(roboRicks);
  }
  public static void sortByState(LinkedList<RoboRick> roboRicks){
          roboRicks.sort((x,y)->x.currentMode.compareTo(y.currentMode));
  }
}

public class ExceptionRoboRick extends Exception {
  public String getMessage() {
     return "exception from robotric";
  }
}
public class ExceptionRoboRickDistance extends ExceptionRoboRickMoving {
  public String getMessage() {
     return "Exception,trying to go too far";
  }
}
public class ExceptionRoboRickNotWheels extends ExceptionRoboRick {
  public String getMessage() {
     return "Exception with wheels";
  }
}
public class ExceptionRoboRickNotEnoughEnergy extends ExceptionRoboRick {
  public String getMessage() {
     return "Exception, not enough energy";
  }
}
public class ExceptionRoboRickMoving extends ExceptionRoboRick {
  public String getMessage() {
     return "Exception on trying to move.";
  }
}
```