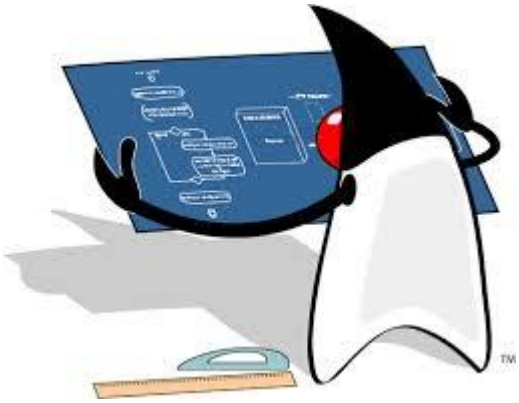
אוניברסיטת בן-גוריון בנגב
Ben-Gurion University of the Negev

משחק מבוך

סמסטר ב' תשע"ט

תקציר

- חלק א' - יצירת ספריית קוד
  - מפסאודו-קוד לתכנות מונחה עצמים
- חלק ב' – שרת-לקוח
  - Streaming, קבצים, ו-decorator
  - תכנות מקבילי עם java threads
- חלק ג'
  - Architectural Design Pattern
  - תכנות מונחה אירועים
  - GUI בטכנולוגיית JavaFX



מחבר: ד"ר אליהו חלסצ'י
khalastc@post.bgu.ac.il

Table of Contents

מנהלות

בקורס זה יינתן תרגיל חימום אחד עם חובת הגשה וללא ציון על מנת להכין אתכם לפרויקט. התרגולים במעבדה נועדו ללמד אתכם את יסודות שפת ***Java***, את מה שיש לה להציע ואת דרכי העבודה המומלצות יחד עם הדגמת החומר הנלמד בהרצאה. הכלים והידע שתלמדו במעבדה נדרשים לכתיבת הפרויקט וימשו אתכם רבות בהמשך דרככם.

לפרויקט שלוש אבני דרך, מועדי הגשתם המעודכנים רשומים באתר ה-Moodle של הקורס.

משקל כל חלק בפרויקט 33% ממשקל הפרויקט הכולל.

כל חלקי העבודה הינם להגשה בזוגות בלבד.

כל חלק בפרויקט נבנה בהתבסס על החלק שקדם לו לכן חשוב לעמוד בדרישות בצורה הטובה ביותר. עמידה בדרישות בחלק מסוים בפרויקט יקל עליכם בחלקים שיבואו אחריו.

הפרויקט מאורגן בצורה מרווחת מאוד כך שיש מספיק זמן לביצוע כל מטלה. כבר ניתנו הזמנים המקסימליים עבור כל מטלה, כל בקשה להארכה קולקטיבית משמעותה לבוא על חשבון המטלות האחרות. להארכות זמן פרטניות מסיבות מוצדקות בלבד יש לפנות למתרגל הקורס.

בכלל זה נוסיף שהעתקה אינה משתלמת; מעבר לסיכון להיתפס ולהישלח (המעתיק והמועתק) לוועדת משמעת, כל צורה של העתקה (גם מפרויקטים דומים קודמים) אינה שווה כלל למימוש קוד עצמאי, ובכך היא פוגעת ביכולת שלכם להפנים את החומר ותוביל לכישלון במבחן.

בהצלחה,

דודי, אביעד ודניאל.

הנחיות כלליות

אנו ממליצים בחום לקרוא את מסמך זה מתחילתו ועד סופו ולהבין את הכיוון הכללי של הפרויקט עוד בטרם התחלתם לממש. הבנת הדרישות ומחשבה מספקת לפני תחילת המימוש תחסוך לכם זמן עקב טעויות מיותרות. *"סוֹף מַעֲשֶׂה בְּמַחְשָׁבָה תְּחִלָּה"*.

עבדו עם ***IntelliJ*** בגירסה העדכנית ביותר, כפי שמותקן במעבדות בהן מועבר בתרגול. את גירסת ה-Community החינמית, תוכלו להוריד מכאן:

https://www.jetbrains.com/idea/download/

עבור כל חלק בפרויקט זה, שעליו אתם מתחילים לעבוד אנו ממליצים:

1. לקרוא את כל הדרישות של החלק מתחילתם ועד סופם.
2. לדון על הדרישות עם בן/בת הזוג למטלה.
3. לחשוב איך אתם הולכים לממש את הדרישות.
4. לייצר לעצמכם ***Class Diagram***.
5. לתכנן את חלוקת העבודה ביניכם.
6. להתחיל לממש.
7. לשמור ולגבות את הגירסאות השונות של הקוד שלכם במהלך העבודה במספר מוקדים שונים: ***Dropbox***, ***Email*** וכו'.
8. לבצע בדיקות עבור כל קוד שמימשתם. בדקו כל מתודה על מנת להבטיח שהיא מבצעת כראוי מה שהיא אמורה לבצע. זכרו שאתם יכולים לדבג (***Debug***) את הקוד שלכם ולראות מה קורה בזמן ריצה.
9. במידה ועשיתם שינויים ו"הרסתם" משהו בפרוייקט, זכרו שב-***IntelliJ*** קיימת האופציה לצפות בהיסטוריה של כל מחלקה ולעקוב אחרי שינויים (מקש ימני על קוד מחלקה > Local History).
10. לשמור את התוצאה הסופית שאותה אתם הולכים להגיש במספר מוקדים שונים.

דגשים לכתיבת קוד:

- הקפדה על עקרונות ה-***SOLID*** שלמדתם.
- הפונקציות צריכות להיות קצרות, עד 30 שורות ולעסוק בעניין אחד בלבד. פונקציות ארוכות ומסובכות שעוסקות בכמה עניינים הם דוגמא לתכנות גרוע.
- הפונקציות צריכות להיות גנריות (כלליות), שאינן תפורות למקרה ספציפי.
- שמות משתנים ברורים ובעלי משמעות.
- שמות שיטות ברורים ובעלי משמעות.
- מתן הרשאות מתאימות למשתנים ולמתודות (***public, protected, private***). כימוס (***Encapsulation***)[1].
- שימוש נכון בתבניות העיצוב שנלמדו בכיתה, בירושה ובממשקים.
- תיעוד הקוד:
  - תיעוד מעל מחלקות, שיטות וממשקים.
  - יש לתעד שורות חשובות בתוך המימוש של השיטות.
  - הסבר על תיעוד Javadoc ניתן למצוא כאן: https://www.tutorialspoint.com/java/java_documentation.htm

בסיום כתיבת הפתרון:

1. הריצו את הפרויקט ובדקו אותו ע"פ הדרישות של החלק אותו מימשתם.

[1] https://he.wikipedia.org/wiki/%D7%9B%D7%99%D7%9E%D7%95%D7%A1

2. עברו שוב על דרישות המטלה ובדקו שלא פספסתם אף דרישה.
3. חשבו על מצבי קצה שאולי עלולים לגרום לאפליקציה שלכם לקרוס וטפלו בהם.

קחו בחשבון שהפרויקט שאתם מגישים נבדק על מחשב אחר מהמחשב שבו כתבתם את הקוד שלכם. לכן, אין להניח שקיים כונן מסוים (לדוגמא *D:*) או תיקיות אחרות בעת ביצוע קריאה וכתיבה מהדיסק.

בנוסף, כאשר אתם כותבים ממשק משתמש, בין אם זה ***Console*** או ***GUI***, קחו בחשבון שהמשתמש (או הבודק) אינו תמיד מבין מה עליו לעשות ולכן עלול לבצע מהלכים "לא הגיוניים" בנסיונו להבין את הממשק שלכם. מהלכים אלו יכולים לגרום לקריסה של הקוד אם לא עשיתם בדיקות על הקלט שהמשתמש הכניס או לא לקחתם בחשבון תרחישים מסוימים. לכן, חשוב שתעזרו למשתמש/בודק לעבוד מול הממשק שלכם ע"י הדפסה של הוראות ברורות מה עליו להקליד, על איזה כפתור ללחוץ (ומתי) ומה האופציות שעומדות בפניו. הנחיות אלו יפחיתו את הסיכויים לטעויות משתמש שעלולות לגרום לקריסה של האפליקציה ולהורדת נקודות.

חלקי הפרויקט שאתם מגישים נבדקים אוטומאטית (פרט לחלק ג' שיבדק גם ידנית) ע"י קוד בדיקה לכן חשוב ביותר להצמד להוראות ולוודא ששמות המחלקות והממשקים שהגדרתם הם <u>בדיוק</u> מה שהתבקשתם. הבדיקה האוטומאטית אינה סלחנית לכן השתדלו להמנע מטעויות מצערות.

לפני הגשת המטלות, בדקו את עצמכם שוב! פעמים רבות שינויים פזיזים של הרגע האחרון שנעשים ללא מחשבה מספקת, גורמים לשגיאות ולהורדת נקודות. חבל לאבד נקודות בגלל טעויות של הרגע האחרון.

הוראות הגשה

את המטלות יש להגיש ל:

1. למערכת הבדיקה האוטומאטית:

http://subsys.ise.bgu.ac.il/submission/login.aspx

*** בחרו סטודנט אחד מתוך הצוות שדרך חשבונו תעלו את העבודות.**
*** למען הסדר הטוב, אתם מתבקשים להעלות את העבודה רק דרך החשבון של הסטודנט הנבחר (ולא מחשבונות שונים בכל פעם).**

**** המטלות יכתבו ב-Java Language Level 8 בלבד.**

הגישו את התוצרים כקובץ ***Zip*** המכיל:

1. תיקיית הפרוייקט שלכם ב-IntelliJ במלואה! התיקיה צריכה להכיל:
   a. תיקיית ***src*** קוד המקור.
   b. תיקיית ה-idea.
   c. תיקיות נוספות כגון resources.
   d. וכו'..
2. קובץ ה-***JAR*** של הפרויקט שלכם.
   a. הוראות יצירה ב-IntelliJ: https://www.youtube.com/watch?v=3Xo6zSBgdgk

- בתיקיה שאתם מגישים מחקו קבצים ותיקיות מיותרות כגון קבצים זמניים או קבצים השייכים ל-Git. (תיקיה נסתרת), התיקיה המוגשת צריכה להיות רזה יחסית.
- במידה ואתם משתמשים בקבצי מדיה, בחרו בקבצים הרזים ביותר (מבחינת גודל קובץ) המספיקים לכם.

קובץ ה-***Zip*** ישא את שמות המגישים בפורמט: ***ID1_ID2.zip***. לדוגמא: ***012345678_123456789.zip***. במידה וברצונכם להעלות גרסה משופרת לפני תום מועד ההגשה תוכלו להעלות קובץ נוסף עם התוספת ***updateX*** כאשר ***X*** הינו מספר העדכון. לדוגמא ***012345678_123456789_update3.zip*** אם העליתם קובץ עדכון שלישי. בעת איסוף העבודות לבדיקה, רק הגרסה האחרונה והעדכנית ביותר תילקח לבדיקה. חשוב שתמיד תעלו קובץ המכיל את ת"ז באותו הסדר.

חשוב להגיש את המטלות בזמן, מטלות שיוגשו לאחר המועד ללא הצדקה לא יבדקו.

הפרויקט המוגש לבדיקה צריך:

- להתקמפל ללא שגיאות. **פתרון שאינו מתקמפל כלל לא ייבדק וציונו יהיה 0.**
- לרוץ ולבצע את מה שהתבקש ללא שגיאות בזמן ריצה. **כל חריגה (Exception) שתגרום לקריסה של האפליקציה בזמן ריצה תגרור הורדת נקודות.**

בדיקת הפרויקט

לקורס קיים בודק תרגילים שיבדוק את חלקי הפרויקט. עם פרסום הציונים יפורסם גם מפתח בדיקה שיפרט את הניקוד עבור כל חלק שנבדק. כפי שפורט קודם לכן, העבודות נבדקות אוטומאטית ע"י קוד בדיקה כך שאין מקום לערעורים מאחר והבדיקה זהה לכולם.

עבור חלקי הפרוייקט א' ו-ב' סופק לכם קוד בדיקה עצמי (Main) שעושה שימוש בחלקים שכתבתם. חשוב שתדאגו, שהקוד שלכם מתקמפל עם ה-Main שסופק. חשוב שלא תעשו שום שינוי ב-Main, אלא רק בקוד שלכם. במידה והפרויקט שלכם אינו מתקמפל במערכת הבדיקה האוטומאטית, נסו שוב, במידה ויש בעיה בקומפילציה במערכת, קיים משהו בקוד שלכם שיוצר את הבעיה ולכן דאגו לפתור את העניין בעצמכם.

במקרים מיוחדים, ניתן להגיש ערעור עד 3 ימים ממועד פרסום הציונים. פניות שיוגשו לאחר מכן לא יבדקו. הגשת ערעור תתבצע <u>במייל בלבד אל בודק התרגילים של הקורס בלבד</u> (לא למתרגלים),ותכיל בכותרת המייל את השמות ות"ז של הסטודנטים. **שימו לב: הגשת ערעור תגרור בדיקה מחודשת ויסודית של העבודה ועלולה אף להוביל להורדה נוספת של נקודות, לכן אל תקלו ראש ותגישו ערעורים על זוטות.**

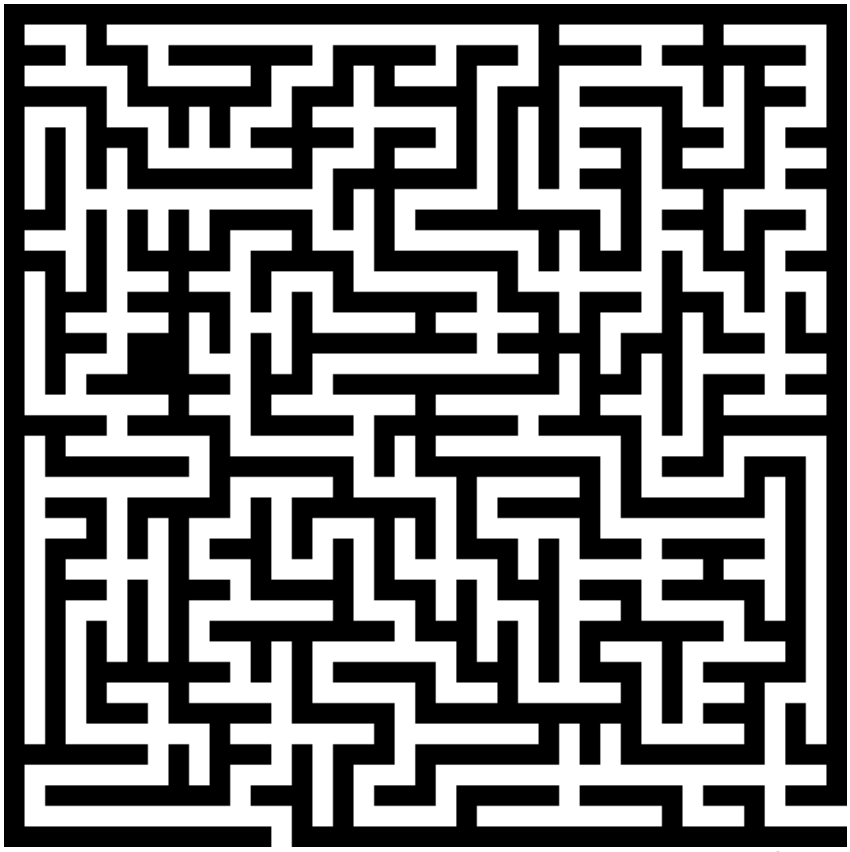
אין אפשרות לבדיקה חוזרת של מטלות. במקרים מיוחדים של טעויות קריטיות שהובילו להורדת ניקוד נרחבת עקב אי-יכולת לבדוק את המטלה תתאפשר בדיקה חוזרת אך יורדו 30 נקודות.

בהצלחה!

הקדמה

מבוך הוא חידה הבנויה ממעברים מתפצלים, אשר על הפותר למצוא נתיב דרכה, מנקודת הכניסה לנקודת היציאה. הדמות יכולה לנוע ימינה, שמאלה, למעלה או למטה, במידה והיעד פנוי מקיר כמובן.

מבוך דו-מימדי



דוגמא למבוך

מבוך דו-מימדי עליכם לייצג כמערך דו-מימדי של int. הערך 1 מסמן תא מלא (קיר) ואילו 0 מסמן תא ריק. נקודת הכניסה לצורך הדוגמא מסומנת באדום ונקודת היציאה בירוק.

```
int[][] maze={
                {0,0,1,0,1,0,0,0,1},
                {1,0,1,0,1,0,1,0,0},
                {1,0,0,0,0,0,0,1,1},
                {1,0,1,1,0,1,0,1,1},
                {0,0,1,1,0,1,0,1,1},
                {1,1,0,0,0,1,0,0,0},
        };
```

המבוך בעל שני מימדים, נקרא להם rows ו-columns המייצגים את מספר השורות והעמודות במבוך.
אין חובה ש-rows=columns, כלומר שהמבוך אינו בהכרח ריבוע, הוא יכול להיות גם מלבן. בנוסף, <u>אין</u> חובה שלמבוך תהיה מסגרת חוסמת ונקודות יציאה מהמסגרת, כפי המופיע בדוגמא.

חלק א': מפסאודו-קוד לתכנות מונחה עצמים

בשיעורים האחרונים למדנו כיצד לתרגם פסאודו-קוד של אלגוריתם לתכנות מונחה עצמים.

למדנו שני כללים חשובים:

כלל ראשון: להפריד את האלגוריתם מהבעיה שאותה הוא פותר.

כשנתבונן בפסאודו-קוד של האלגוריתם נסמן את השורות שהן תלויות בבעיה. שורות אלה יגדירו לנו את הפונקציונליות הנדרשת מהגדרת הבעיה. את הפונקציונליות הזו נגדיר בממשק מיוחד עבור הבעיה הכללית. מאוחר יותר מחלקות קונקרטיות יממשו את הממשק הזה ובכך יגדירו בעיות ספציפיות שונות.

שמירה על כלל זה תאפשר לאלגוריתם לעבוד מול טיפוס הממשק במקום מול טיפוס של מחלקה ספציפית. תכונת הפולימורפיזם תאפשר לנו להחליף מימושים שונים לבעיה מבלי שנצטרך לשנות דבר בקוד של האלגוריתם. **על אילו מעקרונות SOLID שמרנו כאן?**

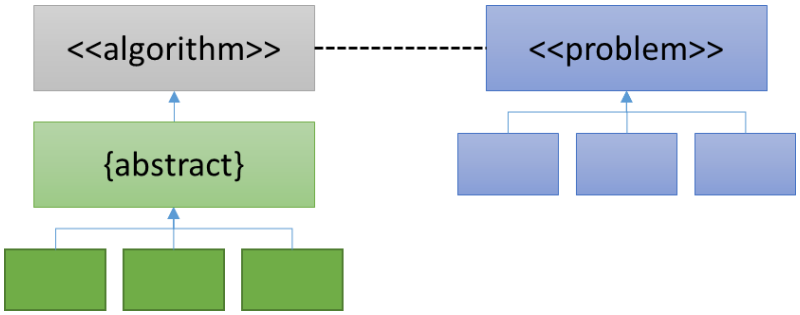
כלל שני: לממש את האלגוריתם באמצעות היררכיית מחלקות.

ייתכנו אלגוריתמים נוספים שנצטרך לממש בעתיד, או מימושים שונים לאותו האלגוריתם שלנו. לכן כבר עכשיו ניצור היררכיית מחלקות שבה

- את הפונקציונלית של האלגוריתם נגדיר בממשק משלו.
- את מה שמשותף למימושים השונים נממש במחלקה אבסטרקטית.
  - את מה שלא משותף – נשאיר אבסטרקטי.
- את המימושים השונים ניצור במחלקות שירשו את המחלקה האבסטרקטית.
  - הם יצטרכו לממש רק את מה ששונה בין האלגוריתמים.

את ההיררכיה הזו ניתן כמובן להרחיב ע"פ הצורך.

קבלנו את המבנה הבא:



על אילו מעקרונות SOLID שמרנו כאן?

משימה א' – אלגוריתם ליצירת מבוך

צרו פרויקט בשם *ATP – Project - PartA* ובתוכו חבילה (Package) בשם algorithms.mazeGenerators. כלומר חבילה בשם algoritgms ובתוכה חבילה בשם mazeGenerators.

בפנים צרו מחלקה בשם Maze המייצגת מבוך כבתיאור לעיל. הוסיפו מתודות למחלקה זו כרצונכם ע"פ הצורך והדרישות בהמשך. מי שהולך ליצור מופעים של Maze יהיה הטיפוס MazeGenerator.

במשימה זו נתרגל את כתיבת ההיררכיה של המחלקות עבור אלגוריתם. את החלק של הבעיה נתרגל בחלק הבא של הפרויקט.

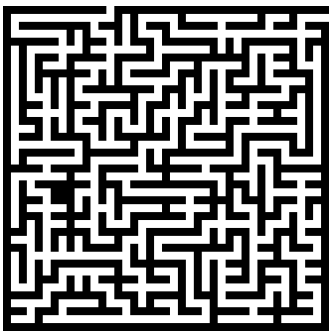
1. הגדירו ממשק בשם IMazeGenerator שמגדיר:
   a. מתודה בשם generate שמחזירה מופע של Maze. המתודה מקבלת שני פרמטרים, מס' שורות ומס' עמודות (כ-int).
   b. מתודה בשם measureAlgorithmTimeMillis מקבלת שני פרמטרים, מס' שורות ומס' עמודות (כ-int), ליצירת מבוך, ומחזירה long.

2. ממשו מחלקה אבסטרקטית כסוג של IMazeGenerator, קראו לה AMazeGenerator.
   a. היא תשאיר את המתודה generate כאבסטרקטית. כל אלג' יממש זאת בעצמו.
   b. לעומת זאת, פעילות מדידת הזמן זהה לכל האלגוריתמים ולמעשה אינה תלויה באלגוריתם עצמו. לכן אותה דווקא כן נממש כאן (במקום לממש אותה כקוד כפול בכל אחת מהמחלקות הקונקרטית).
   c. המתודה measureAlgorithmTimeMillis תדגום את שעון המערכת ע"י System.currentTimeMillis(), תפעיל את generate עם פרמטרים ליצירת מבוך שקיבלה ותדגום את הזמן שוב מיד לאחר מכן. הפרש הזמנים מתאר את הזמן שלקח להפעיל את generate. החזירו את זמן זה כ-long.

3. ממשו מחלקה בשם EmptyMazeGenerator (שתירש את המחלקה האבסטרקטית) שפשוט מייצרת מבוך ריק, חסר קירות.

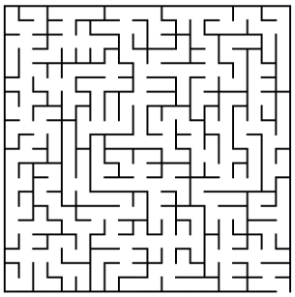
4. ממשו מחלקה בשם SimpleMazeGenerator (שתירש את המחלקה האבסטרקטית) שפשוט מפזרת קירות בצורה אקראית. הבטיחו שלמבוך יש פתרון, קיימות דרכים רבות לעשות זאת.

5. למידה עצמית – עיקר התרגיל. היכנסו לעמוד הבא בוויקפדיה:
   https://en.wikipedia.org/wiki/Maze_generation_algorithm
   בחרו את אחד האלגוריתמים שאתם מתחברים אליו יותר. באמצעותו ממשו מחלקה בשם MyMazeGenerator, היורשת גם היא את המחלקה האבסטרקטית ומייצרת מבוך ע"פ הייצוג לעיל. **דאגו שהאלגוריתם שלכם מייצר מבוכים מעניינים עם מבויים סתומים והתפצלויות כפי הדוגמא:**



6. כתבו את האלגוריתם בצורה יעילה כך שיוכל לייצר מבוך בגודל 1000*1000 בזמן סביר של עד דקה. חשבו על מבני הנתונים הנכונים להשתמש בהם כדי ליעל את האלגוריתם שלכם מבחינת סיבוכיות זמן ומקום.

***** ישנם אלגוריתמים המתייחסים לכל תא במבוך כמוקף ב-4 קירות ובתהליך היצירה הם שוברים את הקירות באופן שמייצר מבוך. מבוכים הנוצרים נראים כך:***



***** תוכלו להבין את האלגוריתמים מסוג זה ולשנות אותם כך שיצרו את המבוך בפורמט המתבקש.***

בדיקות

הוסיפו לפרויקט שלכם Package חדש בשם test. ה-Package יכיל מספר מחלקות הניתנות להרצאה (כוללות פונקציית main) וכל מחלקה תבדוק קטעי קוד אחרים. הוסיפו מחלקה בשם RunMazeGenerator המכילה את הקוד הבא:

```
package test;

import algorithms.mazeGenerators.*;

public class RunMazeGenerator {
    public static void main(String[] args) {
        testMazeGenerator(new EmptyMazeGenerator());
        testMazeGenerator(new SimpleMazeGenerator());
        testMazeGenerator(new MyMazeGenerator());
    }

    private static void testMazeGenerator(IMazeGenerator mazeGenerator) {
        // prints the time it takes the algorithm to run
        System.out.println(String.format("Maze generation time(ms): %s",
mazeGenerator.measureAlgorithmTimeMillis(100/*rows*/,100/*columns*/)));
        // generate another maze
        Maze maze = mazeGenerator.generate(100/*rows*/, 100/*columns*/);

        // prints the maze
        maze.print();

        // get the maze entrance
        Position startPosition = maze.getStartPosition();

        // print the position
        System.out.println(String.format("Start Position: %s",
startPosition)); // format "{row,column}"

        // prints the maze exit position
        System.out.println(String.format("Goal Position: %s",
maze.getGoalPosition()));
    }
}
```

שימו לב ש:

- הקוד נדרש לרוץ במלואו ללא שגיאות.
- המחלקה maze מכילה את השיטות הבאות:
  - getStartPosition – מחזיר את נקודת ההתחלה של המבוך (טיפוס מסוג Position).
  - getGoalPosition – מחזיר את נקודת הסיום של המבוך (טיפוס מסוג Position).
  - את נקודות הכניסה והיציאה מהמבוך אתם קובעים בעת יצירת המבוך.

- Print – מדפיסה את המבוך למסך. סמנו את נקודת הכניסה למבוך בתו **S** ואת נקודת היציאה בתו **E**.

- כחלק ממימוש האלגוריתם שמייצר מבוך, תצטרכו לקבוע למבוך מהי נקודת ההתחלה ומהי נקודת הסיום של המבוך.
- עליכם ליצור מחלקה בשם Position המייצגת מיקום בתוך המבוך. מקמו את המחלקה לצד המחלקה Maze (תחת אותו ה-Package). למחלקה יהיו שני Data Members שייצגו את השורה והעמודה. ההדפסה של Position בקריאה מתוך System.out.println צריכה להחזיר את המיקום בפורמט {row,column}. המחלקה תכיל את המתודות הבאות:
  - getRowIndex()
  - getColumnIndex()
- ה-main בוחן את שני האלגוריתמים, קוד ה-test לא היה צריך להשתנות!

משימה ב' – אלגוריתמי חיפוש

תחת החבילה algorithms צרו חבילה בשם search.

בהתאם לתשתית שראינו בהרצאה:

1. צרו את:
   a. מחלקות: ASearchingAlgorithm, AState, MazeState, Solution.
   b. ממשקים: ISearchable, ISearchingAlgorithm.
2. ממשו את אלגוריתמי החיפוש הבאים:
   a. חיפוש לרוחב Breadth First Search – קראו למחלקה BreadthFirstSearch.
   b. חיפוש לעומק Depth First Search – קראו למחלקה DepthFirstSearch.
   c. אלגוריתם Best First Search – קראו למחלקה BestFirstSearch.
3. כתבו את אלגוריתמי החיפוש כך שיהיו יעילים מבחינת סיבוכיות זמן ומקום. האלגוריתמים צריכים להתמודד עם מבוכים בגודל 1000*1000 בזמן סביר של עד דקה. שימו לב שמבנה המבוך שלכם (תלוי באלגוריתם שיצר אותו) משפיע על כמות המצבים שאלגוריתם החיפוש יצטרך לפתח במהלך החיפוש.
4. צרו **Object Adapter** שמבצע אדפטציה ממבוך (מופע של Maze) לבעיית חיפוש (ISearchable), קראו לו SearchableMaze.
   a. במימוש המתודה getAllPossibleStates תוכלו להחליט אם ממיקום מסויים במבוך נתון להחזיר גם תנועות באלכסון (בנוסף לתנועה אפשריות כמו שמאלה, ימינה, למעלה ולמטה).
   b. תנועה אחת באלכסון מתא X לתא Y אפשרית רק אם ניתן להגיע מתא X ל-Y בשני צעדים רגילים ללא אלכסון (תנועה בצורת 'ר').

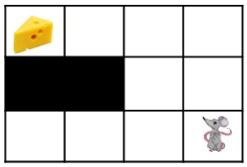
שימו לב ש Best First Search דומה מאד ל Breadth First Search פרט לעובדה שהראשון משתמש בתור עדיפויות (Priority Queue). מומלץ שתור העדיפויות יהיה ממומש כערמה (Heap) לשיפור הביצועים.

ניתן לומר ש Best הוא סוג של ספציפי יותר של Breadth ולכן על המחלקה של Best לרשת את זו של Breadth ולדרוס את התור עם תור עדיפויות. מצד שני, ניתן לומר שמדובר באותו האלגוריתם ובאותו המימוש, פשוט לBreadth First-Search נזין שלכל הקודקודים עדיפות שווה (צעד באלכסון שקול לצעד רגיל). שתי התשובות נכונות.

נשים לב שמספר הקודקודים שכל אלגוריתם ייפתח (מוציא מה open list) הוא שונה. ככל שמפתחים פחות קודקודים כך האלגוריתם יותר יעיל. כדי להבין זאת ולחוש את האלגוריתמים השונים, **לפני המימוש בקוד**, ענו כמה קודקודים ייפתח כל אלג' עבור הדוגמא הבאה:

נתון לנו מבוך שכל תנועה ישרה עולה 10 נקודות, ותנועה באלכסון עולה 15 נקודות (שימו לב שהיא יותר חסכונית משתי תנועות בעלות של 20 המביאות לאותה הנקודה).

נגדיר "מצב" (State) כמיקום של העבר במבוך (עמודה, שורה). במבוך העכבר נמצא ב (2,3). נגדיר שבהנתן מצב, סדר פיתוח השכנים הוא עם כיוון השעון כשמתחילים מלמעלה. כלומר, עבור המצב הנוכחי סדר פיתוח השכנים הוא 1. (1,3) 2. (1,4) 3. (2,4) וכך הלאה עד שנגיע ל (1,2). כמובן שלא נרצה לפתח מצבים שנמצאים מחוץ למבוך או מייצגים קירות. עבור כל אלגוריתם מצאו את מספר הקודקודים שהוא יפתח עד שימצא את המסלול הזול ביותר לגבינה.



בדיקות

תחת ה-Package לבדיקה שיצרתם (test) הוסיפו מחלקה בשם RunSearchOnMaze. צרו במחלקה מתודה Main ש:

a. יוצרת מבוך מורכב באמצעות MyMazeGenerator בגודל 30*30.
b. פותרת אותו באמצעות כל אחד משלושת מהאלגוריתמים.
   i. BreadthFirstSearch
   ii. DepthFirstSearch
   iii. BestFirstSearch
c. עבור כל אלגוריתם:
   i. מדפיסה למסך כמה מצבים פיתח. אם לא מורגש הבדל, הגדילו את המבוך.
   ii. מדפיסה למסך את רצף הצעדים מנקודת ההתחלה לנקודת הסיום.

הריצו את הקוד הבא לדוגמא:

```
package test;

import algorithms.mazeGenerators.IMazeGenerator;
import algorithms.mazeGenerators.Maze;
import algorithms.mazeGenerators.MyMazeGenerator;
import algorithms.search.*;

import java.util.ArrayList;

public class RunSearchOnMaze {
    public static void main(String[] args) {
        IMazeGenerator mg = new MyMazeGenerator();
        Maze maze = mg.generate(30, 30);
        SearchableMaze searchableMaze = new SearchableMaze(maze);

        solveProblem(searchableMaze, new BreadthFirstSearch());
        solveProblem(searchableMaze, new DepthFirstSearch());
        solveProblem(searchableMaze, new BestFirstSearch());
    }

    private static void solveProblem(ISearchable domain, ISearchingAlgorithm
searcher) {
        //Solve a searching problem with a searcher
        Solution solution = searcher.solve(domain);
        System.out.println(String.format("'%s' algorithm - nodes evaluated:
%s", searcher.getName(), searcher.getNumberOfNodesEvaluated()));
        //Printing Solution Path
        System.out.println("Solution path:");
        ArrayList<AState> solutionPath = solution.getSolutionPath();
        for (int i = 0; i < solutionPath.size(); i++) {
            System.out.println(String.format("%s.
%s",i,solutionPath.get(i)));
        }
    }
}
```

- מצופה שאלגוריתם Best First Search ימצא את המסלול הקצר ביותר בין נקודת ההתחלה לנקודת הסיום, מסלול הכולל אלכסונים במידה וניתן.

משימה ג' – Unit Testing (למידה עצמית)

עוד לפני שנתחיל לכתוב את ה GUI נכיר עוד כלי שמאפשר לנו לבדוק את הקוד בפרויקט – Unit Testing.

תפקידינו כמפתחים הוא גם לבדוק את המחלקות שהוא אחראי להן. הוא מתחייב שכל מחלקה שהוא מעלה ל repository היא בדוקה ונמצאה אמינה. מאוחר יותר ה QA בודק האם החלקים השונים של הפרויקט מדברים זה עם זה כמו שצריך ואין בעיות שנוצרות ביניהם.

אחד הכלים המוצלחים נקרא Junit. הרעיון הוא שלכל מחלקה חשובה שכתבנו תהיה לה גם מחלקת JUnit test שבודקת אותה. כך, לאחר שביצענו שינויים בקוד, נריץ תחילה את ריצת הבדיקה, ואם כל הבדיקות "עברו" אז נוכל להריץ את הפרויקט ולהעלות אותו ל repository. במידה ולא עברו, נוכל לפי הבדיקה שכשלה לבודד את התקלה שגרמנו בעקבות השינוי. כך נחסך זמן פיתוח רב.

להלן דוגמא לעבודה עם JUnit ב-Intellij:

https://www.youtube.com/watch?v=Bld3644bIAo

ישנם הגורסים שאת קוד הבדיקות יש לכתוב עוד לפני שכותבים את המחלקה הנבדקת עצמה. כך, הבדיקה תיעשה ללא ההשפעה של הלך המחשבה שהוביל לכתיבת המחלקה, ועלול להיות מוטעה.

עליכם ליצור מחלקת בדיקה לאלגוריתם Best First Search, קראו למחלקה JUnitTestingBestFirstSearch. מקמו את המחלקה בחבילת (package) בדיקה חדשה בשם JUnit. השתמשו ב-JUnit5.

מה בודקים? לא את נכונות האלגוריתם, בשביל זה יש מדעני מחשב. עליכם לבדוק את המימוש של האלגוריתם. כיצד הוא מתנהג עבור פרמטרים שגויים? Null? תבדקו מקרי קצה.

הכל עובד?

מצוין!

משימה ד' – עבודה עם מנהל גרסאות (למידה עצמית)

מעתה אתם עובדים בזוגות. אנו מדמים את המציאות בה אנו מתכנתים כחלק מצוות תכנות. אחד האתגרים הוא ניהול העבודה, ובפרט ניהול הקוד. לא נרצה שתדרסו את הקוד של אחד ע"י השני, או שתלכו לאיבוד בין אינסוף גרסאות ששלחתם במייל... כמו כן, נרצה לשמור גרסאות קודמות כדי שנוכל לחזור לגרסא עובדת במקרה של תקלות או כדי לתמוך במשתמשים בעלי גרסאות קודמות של המוצר שלנו.

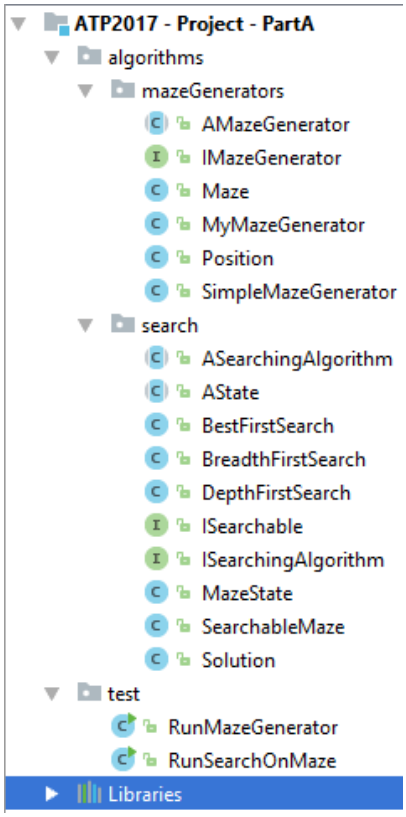
לשם צוותי פיתוח משתמשים במנהל גרסאות.

הרעיון הוא פשוט: בתחילת יום עבודה מורידים ממאגר הקוד שלנו (היושב על שרת כלשהו) את הגרסא האחרונה. עושים את השינויים שלנו על עותק מקומי. לאחר שאנו בטוחים שהשינוי עובד כראוי אנו מעלים אותו חזרה לשרת ומעדכנים את כולם בקוד שלנו. עליכם להתחיל לעבוד כמו המקצוענים.

בפרויקט זה עליכם לעבוד מול GIT. לימדו עצמאית כיצד להגדיר מאגר. יש המון הדרכות וסרטוני הדרכה בנושא ברשת המדגימים את הנושא. כעת צרו פרויקט hello world פשוט ותתנהלו מולו עם שינויים בקוד עד שתבינו כיצד לנהוג כשותפים לאותו מאגר קוד. לאחר שהבנתם כיצד לעבוד יחד תוכלו להתחיל את העבודה על המטלה בפרויקט חדש.

דגשים להגשה

בדקו שהפרויקט שלכם מכיל את החבילות, המחלקות והממשקים בפורמט הבא במדוייק:



***** שימו לב שיש חשיבות לאותיות גדולות וקטנות בטקסט (Case Sensitive).***

**** וודאו שקוד הבדיקה שניתן לכם רץ אצלכם ועובד עם הקוד שלכם כמות שהוא בלי שערכתם בו שינויים.**

בהצלחה!