תרגיל 4: Advanced Java

כללי

1. **מועד ההגשה 9.1.19 בשעה 23:59.**
2. מטרת התרגיל היא היכרות מעמיקה עם תכנות ב- Java, שימוש במנגנוני Reflection, Annotation.
3. קראו היטב את ההוראות, במסמך זה ובקוד שניתן לכם.
4. אחראי על התרגיל: **אריק**. שאלות יש לשלוח למתרגל האחראי על התרגיל במייל eric.oop.course@gmail.com **עם הנושא: "HW4 236703"**. **שאלות בנושאים אדמיניסטרטיביים יש לשלוח לנתן.**
5. הקפידו על קוד ברור, קריא ומתועד ברמה סבירה. עליכם לתעד כל חלק שאינו טריוויאלי בקוד שלכם.
6. מהירות ביצוע אינה נושא מרכזי בתרגילי הבית בקורס. בכל מקרה של התלבטות בין פשטות לבין ביצועים, העדיפו את המימוש הפשוט.
7. הימנעו משכפול קוד והשתמשו במידת האפשר בקוד שכבר מימשתם.
8. העדיפו שימוש **בתכונות מתקדמות שהוצגו בגרסה 8 של Java** על שימוש בתכונות ישנות יותר (למשל: העדיפו שימוש במתודת forEach של Stream/Collection על שימוש בלולאת (:)for).
9. בכדי להימנע מטעויות, אנא עיינו ברשימת ה FAQ המתפרסמת באתר באופן שוטף.
10. **אנו ממליצים להתחיל לפתור את התרגיל מוקדם ככל האפשר, משום שהוא ארוך מקודמיו.**

הקדמה

בתרגול הוזכרה בקצרה חבילת העזר SUnit לשפת Smalltalk וכן חבילת JUnit המקבילה לה בשפת Java. ספריות אלה מממשות את עקרון בדיקות היחידה (Unit Testing) שלקח תפקיד חשוב בהתקדמות של פיתוח מונחה בדיקות (Test driven development). בדיקות יחידה הן בדיקות ברמת יחידת המערכת הקטנה ביותר (לרוב מחלקה) שמאמתות את פעילותה התקינה של היחידה. הבדיקות נערכות לרוב לאחר הכנת המחלקה או לאחר שבוצעו בו שינויים, אך קיימות גם שיטות בהן מורצות בדיקות יחידה בטרם הכנת המחלקה. הרעיון הוא ליצור קבוצה של בדיקות, אשר תכסה את כל פעילות המחלקה, והצלחתה תוכיח בוודאות סבירה כי המחלקה תקינה. בדיקות יחידה לתוכנה היו בשימוש מאז ימיו הראשונים של התכנות הפרוצדוראלי. המתכנת היה כותב לעתים תוכנית בדיקה שאינה חלק מהיישום על מנת לבדוק נכונות של פרוצדורה. השיטה הייתה ישימה גם בתכנות מודולארי ובכל פרדיגמת תכנות נפוצה מאז, אך חסרונה המרכזי היה נעוץ בקושי לערוך סדרת בדיקות מקיפה המורצת באופן אוטומטי.

חל מהעשור הראשון של המאה ה-21 גבר השימוש בחבילות עזר לבדיקות יחידה (Unit testing frameworks) שאפשרו יצירת תבנית אחידה לכל בדיקת יחידה. ספריות עזר נוספות אפשרו את הגברת הבידוד של כל בדיקה והרצתה בהקשר עצמאי לחלוטין, ללא תלות ביחידות אחרות או בתלויות חיצוניות אחרות דוגמת בסיס נתונים.

בניית בדיקות היחידה באמצעות חבילות העזר מאפשרת אוטומציה מלאה של ההרצה שלהן.

בדיקת היחידה נכתבת באמצעות שרותי ספרייה של החבילה המאפשרים לסמן האם הבדיקה הצליחה או נכשלה. כך ניתן להריץ סדרה של בדיקות ולוודא שכולן מצליחות, או לסירוגין – לבדוק מדוע בדיקה נכשלת, ולתקן את הרכיב בהתאם. כך יכולים מתכנתים להריץ בפקודה אחת סדרה של בדיקות השייכות למחלקות הקשורות להם.

בתרגיל זה תממשו חבילת Unit Testing בשפת התכנות Java ע"י כתיבת מחלקה חדשה בשם OOPUnitCore שתפקידה להריץ את הבדיקות ולאסוף את התוצאות.

חלק 1 – הגדרת אנוטציות וטיפוסי עזר

עליכם להגדיר שש אנוטציות: OOPExceptionRule, OOPTestClass, OOPSetup, OOPBefore, OOPAfter ו- OOPTest. על כל אנוטציה להיות מוגדרת באמצעות המטה-אנוטציות Retention ו- Target המתאימות (על ההתאמה להיות **מדויקת**, בהתאם לשימוש באנוטציה). בנוסף, לחלק מהאנוטציות נגדיר טיפוסי עזר.

OOPTestClass

אנוטציה המוצמדת ל**טיפוס** כלשהו **בלבד.** המטרה של OOPTestClass היא לסמן שמחלקה מסוימת כוללת בתוכה בדיקות יחידה. מחלקה בעלת אנוטציה זו תיקרא מחלקת בדיקה. לאנוטציה זו תכונה אחת מטיפוס OOPTestClassType (ראה בהמשך) בשם value. ערך ORDERED מציין שיש להריץ את הטסטים בסדר מסוים שיוגדר ע"י האנוטציה OOPTest, ערך UNORDERED מציין כי אין חשיבות לסדר הרצת הטסטים. כברירת מחדל אין סדר בהרצת הטסטים.

- **OOPTestClassType**

מחלקת enum בעלת שני מופעים : *ORDERED, UNORDERED*.

על enum זה להיות **מקונן** בתוך OOPTestClass ולהיות בעל רמת הרשאה נמוכה ככל הניתן.

OOPSetup

אנוטציה המשמשת לסימון **מתודות בלבד**. מתודה המסומנת באנוטציה זו תורץ פעם אחת בלבד, טרם הרצת הבדיקות. מטרת המתודה היא אתחול שדות הזקוקים לאתחול פעם אחת בלבד (משמש בעיקר לפעולות "כבדות" כמו אתחול מסד נתונים וכו'). מתודה המסומנת בעזרת האנוטציה הנ"ל תיקרא **מתודת אתחול**.

OOPBefore

אנוטציה המשמשת לסימון **מתודות בלבד.** מתודה המסומנת באנוטציה זו תורץ לפני הרצת מתודות הבדיקה ששמן מופיע במערך המוחזק כאלמנט. מטרת המתודה היא ליצור אי תלות בין הבדיקות עצמן (משמש בעיקר לאתחול שדה, שאחד מהטסטים שירוץ אחרי מתודה זו, קורא ממנו, על מנת לוודא כי הערך לא מושפע

מבדיקות שרצו קודם). על האנוטציה להכיל שדה בשם value מסוג String[] שיחזיק את רשימת שמות המתודות שצריך להריץ לפניהן את המתודה הזאת.

OOPAfter

אנוטציה המשמשת לסימון **מתודות בלבד**. מתודה המסומנת באנוטציה זו תורץ אחרי הרצת מתודות הבדיקה ששמן מופיע במערך המוחזק כאלמנט. מטרת האנוטציה היא לאפשר שחרור של משאבים שהוקצו במהלך הריצה של הטסטים. על האנוטציה להכיל שדה בשם value מסוג String[] שיחזיק את רשימת שמות המתודות שצריך להריץ אחריהן את המתודה הזאת.

OOPTest

אנוטציה המשמשת לסימון מתודות. משמעות האנוטציה היא שהמתודה המסומנת היא מתודת בדיקה. לאנוטציה יהיה שדה order מטיפוס int, המציינת את הסדר שבו יש להריץ את מתודות הבדיקה. שדה זה מהווה את הסדר עבור מחלקות בדיקה המסומנות בעזרת ORDERED.

ניתן להניח כי הסדרים רציפים (כלומר לא תהיה מתודה המסומנת ב 3, מבלי שתהיינה מתודות המסומנות ב 1,2) ומתחילים ב-1.

בנוסף, יהיה לה שדה נוסף בשם tag מטיפוס מחרוזת, שמשמעותה היא תיוג המתודה לצורך הרצה סלקטיבית של מתודות בדיקה (יפורט בהמשך). הערך הדיפולטי של תכונה זו היא מחרוזת ריקה – "".

בצורה הדיפולטית, מתודות הבדיקה לא אמורות לזרוק חריגות כדי שהן יצליחו.

OOPExceptionRule

אנוטציה המשמשת לסימון שדות בלבד. מטרת האנוטציה היא לסמן "חוק" הנוגע לחריגות, שביחד עם טיפוס שיוגדר בהמשך בשם OOPExpectedException, יקבע האם צפויה להיזרק חריגה במהלך מתודת בדיקה. לצורך פשטות, האנוטציה תסמן אך ורק שדות מטיפוס OOPExpectedException.

הבהרה: אנוטציה זו מסמנת שדה המגדיר איך להתייחס לחריגות, וחריגות מתגלות בזמן ריצה, לכן על האנטוציה להיות זמינה בזמן ריצה.

OOPExpectedException

מנשק שמגדיר פרוטוקול עבור יצירת אובייקט למחלקת בדיקות אשר יהווה בקרה על הנושא של שגיאות וחריגות של מתודת בדיקה. לצורך כך, נתון לכם קובץ המנשק בשם OOPExpectedException, אשר מגדיר בתוכו את הפרוטוקול הנ"ל הצפוי משדה מטיפוס זה. עליכם להגדיר מחלקה בשם OOPExpectedExceptionImpl, אשר תממש את המנשק, ותגדיר את ההתנהגות הרצויה. דוגמה לשימוש בשדה מטיפוס זה ניתן לראות בטסט לדוגמה שמסופק ע"י הצוות.

<u>עליכם להגדיר את ה- Annotations בדיוק כפי שהן מתוארות לעיל!</u>

חלק 2 – מחלקת התוצאות

OOPResult 2.1

כדי לייצג תוצאה של מתודת בדיקה יחידה, נשתמש ב- interface בשם OOPResult **(המנשק נמצא בקוד המסופק יחד עם התרגיל).**

בתוך OOPResult מוגדר enum בשם OOPTestResult . ל- OOPTestResult ישנם ארבעה מופעים:

SUCCESS, FAILURE, ERROR, EXPECTED_EXCEPTION_MISMATCH

אתם מתבקשים לממש את OOPResult במחלקה משלכם בשם OOPResultImpl ולממש בה את כל המתודות הבאות:

- public OOPTestResult getResultType()

  מתודה זו מחזירה את ה- enum שתואר לעיל ,המתאר את תוצאת מתודת הבדיקה.
- public String getMessage()

  מתודה זו מחזירה הודעה המתאימה לתוצאת הבדיקה.
- public boolean equals(Object obj)

  עליכם לממש מתודת equals. ההשוואה צריכה להתבצע על פי ערכי החזרה של שתי המתודות getResultType() ו- getMessage().

כיצד יודעים מהי תוצאת הבדיקה?

כאשר מריצים מתודת בדיקה, נאמר שמתודת הבדיקה **הצליחה** אם לא נזרקה אף חריגה כתוצאה מהרצת מתודת הבדיקה או אם נזרקה החריגה עם הודעת שגיאה מתאימה שציפינו לה באמצעות שדה OOPExpectedException בתוך מחלקת הבדיקות. נאמר כי מתודת טסט מצפה לחריגה אם היא נמצאת במחלקה שמכילה שדה (לא סטטי) מטיפוס OOPExpectedException ונקרא עליו מתודת expect **בתוך הטסט**. במקרה כזה, אובייקט הממשמש OOPResult המתאר את תוצאות מתודת הבדיקה יכיל OOPTestResult.**SUCCESS** וההודעה המתאימה לתוצאת הבדיקה היא null.

אם נזרקה חריגה (לא מתאימה) כתוצאה מהרצת מתודת הבדיקה, יש לבדוק מהי החריגה שנזרקה:

- אם נזרקה חריגה מטיפוס OOPAssertionFailure , אובייקט הממשמש OOPResult המתאר את תוצאות מתודת הבדיקה יכיל OOPTestResult.**FAILURE** וההודעה המתאימה תהיה הודעת החריגה (הניתנת ע"י getMessage()).
- אם נזרקה חריגה, אך אין זו חריגת OOPAssertionFailure, **ולא ציפינו לחריגה מטיפוס זה, <u>או</u> שציפינו לחריגה מסוג זה, אך ציפינו להודעת שגיאה מסוימת שלא הופיעה**, אז אובייקט הממשמש OOPResult המתאר את תוצאות מתודת הבדיקה יכיל OOPTestResult.**EXPECTED_EXCEPTION_MISMATCH** וההודעה המתאימה תהיה <u>ההודעה שחוזרת מ-getMessage() של הטיפוס OOPExceptionMismatch שמסופק לכם</u>.

- אחרת, אם נזרקה חריגה, ולא ציפינו בכלל שתיזרק חריגה, או לחלופין ציפינו שתיזרק חריגה ולא נזרקה אחת, אז אובייקט הממש OOPResult המתאר את תוצאות מתודת הבדיקה יכיל OOPTestResult.**ERROR** וההודעה המתאימה תהיה שם המחלקה של החריגה.

דוגמה:

אם הרצנו מתודת בדיקה, וכתוצאה מכך נזרקה חריגה e מסוג OOPAssertionError, ההודעה המתאימה תהיה המחרוזת המוחזרת על- ידי e.getMessage(). אם הרצנו מתודת בדיקה, וכתוצאה מכך נזרקה חריגה e מטיפוס A, כאשר ציפינו לחריגה מטיפוס B, והם לא תואמים, ההודעה המתאימה תהיה המחרוזת המוחזרת על-ידי new OOPExceptionMismatchError(A, B).getMessage().

OOPTestSummary 2.2

המחלקה OOPTestSummary משמשת להגדרת תוצאות של הרצת מספר מתודות בדיקה.

OOPTestSummary מורכבת ממיפוי אשר מתאר עבור כל מתודה מהי תוצאת הבדיקה שלה. מיפוי זה הנו מיפוי משם המתודה אל אובייקט הממש OOPResult, המתאר את תוצאות הבדיקה שלה.

עליכם להגדיר את המחלקה OOPTestSummary אשר תומכת בפונקציות הבאות:

- OOPTestSummary(Map<String, OOPResult> testMap)

  בנאי המקבל מיפוי בין שם מתודת הבדיקה לבין פירוט תוצאת הפעלתה.
- int getNumSuccesses()

  מחזירה את סה"כ מספר הבדיקות שהצליחו.
- int getNumFailures()

  מחזירה את סה"כ מספר הבדיקות אשר נכשלו בגלל חריגת OOPAssertionError.
- int getNumExceptionMismatches()

  מחזירה את סה"כ מספר הבדיקות אשר נכשלו בגלל זריקת חריגה לא תואמת את הצפוי.
- int getNumErrors()

  מחזירה את סה"כ מספר הבדיקות אשר נכשלו בגלל זריקת חריגה אחרת.

חלק 3 – הגדרת OOPUnitCore

בחלק זה, תממשו את המחלקה העיקרית OOPUnitCore. מחלקה זו מכילה מתודות סטטיות, אשר מספקות את הפונקציונליות הדרושה להרצה של בדיקות יחידה.

הגדירו את המחלקה OOPUnitCore, וממשו בה את המתודות הבאות:

- `void assertEquals(Object expected, Object actual)`

מתודה סטטית שתפקידה לוודא שאכן קיבלה שני אובייקטים שווי תוכן. אם לא, יש לזרוק OOPAssertionFailure (המסופקת לכם בתרגיל).

- `void fail()`

מתודה סטטית שתפקידה להכשיל מתודת בדיקה ע"י זריקת חריגת OOPAssertionFailure.

- `OOPTestSummary runClass(Class<?> testClass)`
- `OOPTestSummary runClass(Class<?> testClass, String tag)`

- מתודות אלו (runClass) מקבלות אובייקט מחלקה (ואולי tag). אם הפרמטר הוא null, או שהמחלקה אינה מחלקת בדיקה (אין לה את ה-Annotation שהגדרנו, OOPTestClass) יש לזרוק חריגת IllegalArgumentException.

מה עושה runClass?

א. המתודה תיצור מופע חדש של המחלקה. ניתן להניח שקיים במחלקה בנאי חסר ארגומנטים (לא בהכרח פומבי).

ב. המתודה תריץ את **כל** מתודות האתחול (OOPSetup) במחלקה ובמחלקות האב שלה. האובייקט שעליו תופעלנה המתודות יהיה זה שנוצר בשלב א.

ג. המתודה תריץ את **כל*** מתודות הבדיקה במחלקה, ובמחלקות האב שלה (*אשר מתויגות עם תגית אשר מכילה את ה- tag שנשלח ל- runClass, אם נשלח כזה). האובייקט שעליו תופעלנה המתודות יהיה זה שנוצר בשלב א'.

ד. לפני כל הרצה של מתודת בדיקה כלשהי, על המתודה לבצע הרצה של **כל** המתודות במחלקה ובמחלקות האב שלה בעלות האנוטציה OOPBefore, אשר שם מתודת הבדיקה מופיע בערך value של האנוטציה OOPBefore, כפי שהוגדר לעיל.

ה. אחרי הרצה של מתודת בדיקה כלשהי, על המתודה לבצע הרצה של **כל** המתודות במחלקה ובמחלקות האב שלה בעלות הסימון OOPAfter, אשר שם מתודת הבדיקה מופיע בערך value של האנוטציה OOPAfter, כפי שהוגדר לעיל.

ו. המתודה תחזיר אובייקט מטיפוס OOPTestSummary המתאר את תוצאות הרצת מתודות הבדיקה, כפי שהוגדר לעיל.

הערות

- בעת הרצת runClass, יש להתייחס **לכל** הטסטים בעץ הירושה כ-ORDERED/UNORDERED בהתאם לקלאס שהועבר לפונקציה.

- במקרה שמורצת מחלקה ORDERED עם אב קדמון שהוא UNORDERED, יש להתייחס לטסטים של האב כטסטים עם order שווה אפס.

גיבוי מצב האובייקט ושחזורו במידת הצורך

כדי למנוע מצב שבו מתודות OOPBefore\OOPAfter זורקות חריגות ואז מביאות את האובייקט למצב לא תקין, נבצע **גיבוי לשדות האובייקט** לפני הפעלת המתודה (לפני ה- before) ונשחזר במידת הצורך.

לשם הפשטות, הגיבוי צריך להתבצע על השדות של מחלקת אובייקט היעד בלבד, ולא על השדות של המחלקות במעלה שרשרת הירושה. כדי להגדיל את הסיכוי שהגיבוי יחזיק ערכים שלא יושפעו מהפעלת המתודה, עליו להתבצע לפי סדר העדיפויות הבא:

1. אם האובייקט שבשדה תומך ב- clone, הגיבוי ישמור שכפול של אותו האובייקט.
2. אם לאובייקט יש copy constructor (בנאי המקבל פרמטר יחיד שהינו אובייקט מאותו הטיפוס), יעשה בו שימוש כדי ליצור אובייקט חדש מאותו הטיפוס, והוא זה שישמר בגיבוי.
3. אם שתי האפשרויות הקודמות לא קיימות, יישמר בגיבוי האובייקט שבשדה עצמו.

לדוגמא, אם באובייקט שעליו מפעילים את המתודה הנבדקת יש שדה מטיפוס java.util.Date (שמממש את Cloneable), נשמור בגיבוי עותק שלו שניצור באמצעות קריאה ל- clone. אם יהיה שדה מטיפוס String, נשתמש בבנאי שלו שמקבל String.
התוצאה של שחזור הגיבוי היא שכל השדות העומדים בשני התנאים הראשונים יצביעו על אובייקטים שונים מאלה שהצביעו לפני הרצת המתודה, אבל ה- state של האובייקטים המוצבעים לפני ואחרי יהיה זהה.
במקרה שפונקציה שמסומנת ע"י OOPBefore זורקת חריגה יש לשחזר את מצב האובייקט כפי שהיה לפני הפעלתה ולהמשיך לבצע את הטסט הבא. אותו דבר יש לעשות במקרה ש- OOPAfter זורקת חריגה (לשחזר את האובייקט כפי שהיה לפני הפעלת פונקציית ה-after). במקרה של זריקת חריגה מתוך מתודות after\before, תוצאת הבדיקה צריכה להיות OOPTestResult.ERROR.

דרישות והנחות נוספות

- בהקשר של ירושה, יש להריץ את פונקציות ה- setup של כל אחת מהמחלקות, החל מהאב הכי עליון ועד הבן. את פונקציות ה- before יש להריץ קודם גם בסדר זה (מהאב העליון ועד הבן). לעומת זאת, את פונקציות ה- after יש להריץ בסדר הפוך, כלומר מהמחלקה הנוכחית ועד האב העליון (חשבו על זה כעל סדר הרצת בנאים והורסים, כפי שנלמד בתרגול).
- אם מתודה f המסומנת ב-@OOPSetup המוגדרת במחלקה X, נדרסת במחלקה Y היורשת מ- X, אין להפעיל את f של X, אלא רק את f של Y.

- אין חשיבות לסדר הפעלת פונקציות ה-OOPBefore שנמצאות באותה מחלקה. כנ"ל לגבי OOPAfter.
- אם פונקציית OOPBefore או OOPAfter נכשלת, תוצאת הבדיקה תהיה **OOPTestResult.ERROR**. ההודעה המתאימה תהיה שם המחלקה של החריגה.
- ניתן להניח שכל המתודות במחלקת הבדיקה אינן מקבלות פרמטרים ובעלות טיפוס החזרה void.
- ניתן להניח שהמתודות OOPSetup לא תזרוק חריגות במהלך הפעלתן.
- שימו לב שמתודת OOPBefore יכולה לרוץ מספר פעמים, למעשה תרוץ פעם אחת לפני כל הפעלה של טסט שמופיע ברשימה שלה. כנ"ל לגבי OOPAfter.
- ניתן להניח כי כל מתודה מסומנת בלכל היותר אנוטציה אחת מאלה שנורשו.
- ניתן להניח כי תהיה מתודה אחת לכל היותר המוגדרת במחלקה ומסומנת באנוטציה OOPSetup.
- ניתן להניח כי יהיה לכל היותר שדה אחד שמסומן בעזרת @OOPExceptionRule, וכן כי אם יש שדה מטיפוס OOPExpectedException, הוא יהיה מסומן עם האנוטציה הזו.
- שימו לב כי יש צורך לבצע איפוס לשדה מטיפוס OOPExpectedException בין טסטים. המצב הדיפולטי של טסט הוא שהוא לא מצפה לאף חריגה (מחלקה או הודעה).

הערות כלליות – חשוב!!!

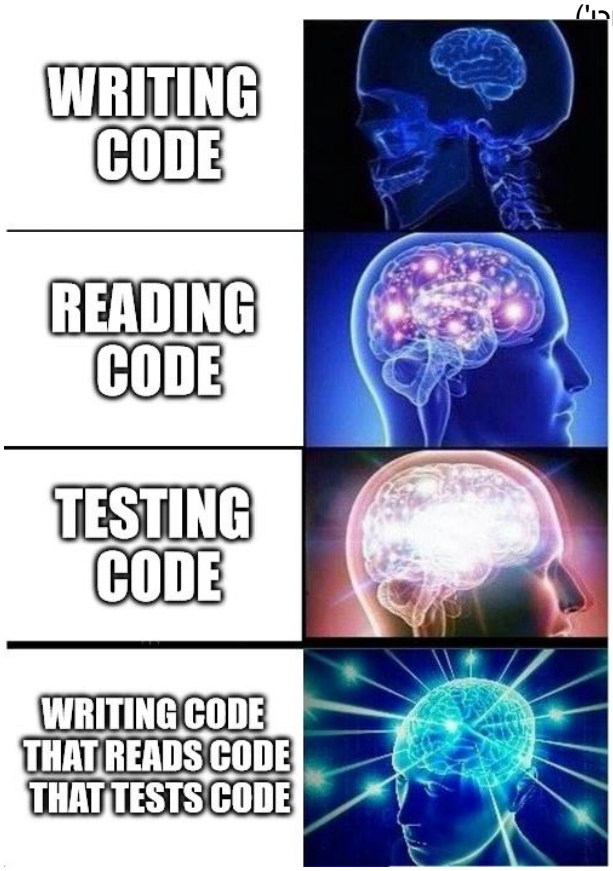
- **במקומות בהם הוגדרו שמות בתיאור ובקבצים, יש להשתמש בשמות אלה במדויק**. הבדיקה תתבצע בצורה אוטומטית, וכישלון בהידור עקב שימוש בשמות לא נכונים יביא להורדת נקודות.
- הימנעו משכפול קוד. פעולות מסוימות חוזרות על עצמן, וניתן לבצע אותן במתודה נפרדת אחת.
- ממשק ה-Reflection של ג'אווה הוא פשוט וידידותי. תמצאו בו getters כמעט לכל סוג של מידע שתרצו לקבל. היעזרו בהשלמה האוטומטית של ה IDE החביב עליכם (IntelliJ!) כדי לדעת אילו אפשרויות יש לכם בכל נקודה.
- מומלץ להשתמש ב-JUnit כדי לבדוק את הפתרון שלכם – בבדיקות שמסופקות יש גם הדגמה של בדיקת זריקה של חריגה. עם זאת, אין חובה להשתמש ב-JUnit, ואין צורך להגיש את הבדיקות. כאשר תגיעו למצב בו אתם בטוחים בפתרון שלכם, תוכלו לנסות להגדיר, למען השעשוע, טסטים באמצעות המנגנון שכתבתם, ולבדוק את המנגנון שלכם בעזרת המנגנון עצמו.
- **כל הקבצים המסופקים נחשבים לחלק מתיעוד התרגיל – עליכם האחריות לקרוא אותם ולהבין את תפקיד כל אחד מהם.**

הוראות הגשה

- **בקשות לדחייה, מכל סיבה שהיא, יש לשלוח למתרגל האחראי על הקורס (נתן) בלבד. שימו לב שבקורס יש מדיניות איחורים, כלומר ניתן להגיש באיחור גם בלי אישור דחייה – פרטים באתר הקורס תחת General info.**
- הגשת התרגיל תתבצע אלקטרונית בלבד (יש לשמור את אישור השליחה!)
- יש להגיש קובץ בשם OOP4_<ID1>_<ID2>.zip המכיל:
  - קובץ בשם readme.txt בפורמט הבא:

```
name1 id1 email1
name2 id2 email2
```

- הקוד: כל קבצי הקוד שלכם שנמצאים בחבילה **Solution** (ללא התיקייה עצמה).
  במילים אחרות, כל קבצי הקוד אמורים להימצא ישירות בתוך ה-zip, ולא בתוך תיקיה.
- **הימנעו** משימוש בתיקיות בתוך ה-zip ומהגשת קבצים שבחבילות האחרות.
- נקודות יורדו למי שלא יעמוד בדרישות ההגשה (rar במקום zip, קבצים מיותרים נוספים, readme בעל שם לא נכון וכו')



בהצלחה!