תוכנה 1

תרגול 3: מבוא לתכנות באמצעות חולית

סיפור דמיוני של NASA ו-Microsoft

- NASA פנתה לחברת Microsoft בכדי שתפתח עבורה מערכת הפעלה לגשש המיועד להישלח לכוכב הלכת מאדים. בעת חישוב מסלול הנחיתה של הגשש על מאדים, בוצעה קריאה לאחת המתודות של מערכת ההפעלה שהובילה להתרסקותה. כתוצאה מכך התרסק גם הגשש על מאדים. NASA האשימה מייד את חברת Microsoft בהתרסקות.
- שאלה: האם NASA צודקת?
- תשובה: יש לבדוק את החוזה בין NASA ל- Microsoft:
  - האם אופן הקריאה למתודה (פרמטרים) היה תקין ע"פ החוזה?
  - האם התגובה של לקריאה למתודה היא חוקית ע"פ החוזה?

המשך הסיפור הדמיוני: "הבג שלנו הבג שלה"

- NASA הגישה תביעה של 100M$ נגד Microsoft.
- בכתב ההגנה שהציגה חברת Microsoft נטען:
  - התעופה ארעה בקריאה למתודה
    `double min(double[] arr)`
    המחזירה את ערך המינימום במערך הנתון.
  - הארגומנט שהועבר למתודה היה null (arr == null)
  - במפרט של המתודה מצוין כי arr =! null.
- סוף דבר: NASA הפסידה בתביעה.

חוזה בין ספק ללקוח

- בספור הדמיוני שהוצג:
  - חברת Microsoft : ספקית של שרות.
  - NASA : לקוחה של שרות.
  - השרות: מערכת הפעלה לגשש.
- חוזה בין ספק ללקוח מגדיר עבור כל שרות:
  - תנאי ללקוח - ידוע בשם "תנאי קדם" (precondition).
  - תנאי לספק - ידוע בשם "תנאי אחר" – postcondition.

תנאי קדם (preconditions)

- מגדירים את הנחות הספק
- ברוב המקרים, ההנחות הללו מתארות מצבים של התוכנית שבהם מותר לקרוא לספק
- במקרים פשוטים (ונפוצים), ההנחות הללו נוגעות רק לקלט שמועבר לשירות.
- במקרה הכללי ההנחות הללו מתייחסות גם <u>למצב</u> התוכנית, כגון משתנים גלובליים.
- תנאי הקדם יכול להיות מורכב ממספר תנאים שעל כולם להתקיים (AND)

תנאי אחר (postconditions)

- מגדיר את המחוייבות של הספק
- אם תנאי הקדם מתקיים, הספק חייב לקיים את תנאי האחר
- ואם תנאי קדם אינו מתקיים? לא ניתן להניח דבר:
  - אולי השרות יסתיים ללא בעיה
  - אולי השרות יתקע בלולאה אינסופית
  - אולי התוכנית תעוף מייד
  - אולי יוחזר ערך שגוי
  - אולי השרות יסתיים ללא בעיה אך והתוכנית תעוף / תתקע לאחר מכן
  - ...
- ובכתיב לוגי: תנאי קדם $\Leftarrow$ תנאי אחר,
  (תנאי קדם)! $\Leftarrow$?

min1 פונקציה

```
/*
 * precondition:  1) arr != null
 *                2) arr.length > 0
 *                3) arr contains only numbers (no NaN or ±infinity)
 *
 * postcondition: Returns the minimal element in arr
 */
public static double min1(double[] arr) {
        double m = 1.0/0.0; // Infinity
        for (double x: arr)
                m = (x < m ? x : m);
        return m;
}
```

המימוש אינו בודק את קיומם של תנאי הקדם

מה יקרה אם בקריאה ל- min1 לא יקויימו כל התנאים בתנאי הקדם?

arr==null?

arr.length == 0?

arr מכיל NaN?

arr מכיל Infinity או Infinity–?

דוגמא 2 (אותו קוד, תוצאה שונה)

```
/*
 * precondition:  arr != null
 *
 * postcondition:
 * If ((arr.length==0) || (arr contains only NaNs)) returns Infinity.
 * Otherwise,  returns the minimal value in arr.
 */
 public static double min2(double[] arr) {
        double m = 1.0/0.0; // Infinity
        for (double x: arr)
                m = (x < m ? x : m);
        return m;
}
```

בהשוואה לחוזה מדוגמא 1:
חוזה מתירני יותר מבחינת הלקוח

דוגמא 3 (טיפול שונה ב- NaN)

```
/*
 * precondition:  arr != null
 *
 * postcondition: If (arr.length=0) returns Infinity.
 * Otherwise,  if arr contains any NaN – returns NaN.
 * Otherwise,  returns the minimal value in arr.
 */
 public static double min3(double[] arr) {
        double m = 1.0/0.0; // Infinity
        for (double x: arr){
                if (Double.isNaN(x)) return x;
                m = (x < m ? x : m);
        }
        return m;
}
```

השוואה לחוזה מדוגמא 2:
טיפול שונה במקרה קצה (קיום ערכי NaN)

(precondition אאא) 4 אאבוד

תמיד מתקיים

תנאי אחר המגדיר תגובה לכל קלט אפשרי מסבך את הקוד.

```
/*
 * precondition:  none
 *
 * postcondition: If ((arr==null) || (arr.length==0)) returns NaN
 * Otherwise,  if arr contains only NaN – returns Infinity.
 * Otherwise,  returns the minimal value in arr, ignoring any NaN.
 */
 public static double min4(double[] arr) {
        if ( (arr==null) || (arr.length==0)) return Double.Nan;
        double m = 1.0/0.0; // Infinity
        for (double x: arr){
                m = (x < m ? x : m);
        }
        return m;
}
```

(precondition ללא) 5 מקרה

```
/*
 * precondition:  none
 *
 * postcondition: If ((arr != null) && (arr.length> 0) &&
 * (arr contains only numbers)) - returns the minimal value in arr.
 * Otherwise,  the return value is undefined.
 */
 public static double min5(double[] arr) {
          if (arr==null) return 0;
          double m = 1.0/0.0; // Infinity
          for (double x: arr)
                    m = (x < m ? x : m);
          return m;
}
```

תנאי אחר המגדיר תגובה רק לקלט פשוט. עבור קלטים אחרים - מתחייב להחזיר ערך כלשהו לא מוגדר (כלומר לסיים קריאה באופן תקין)

הוכחת נכונות של שיטות

- חוזים עוזרים בהוכחת הנכונות של המימוש. כלומר שהמימוש מקיים את החוזה.
- הוכחת נכונות פורמלית של מגבירה את בטחוננו בנכונות המימוש. חסרונה – דורשת מאמץ ניכר (עלות גבוהה של כ"א).
- מפתחי תוכנה מנוסים מבצעים באופן אינטואיטיבי הוכחות נכונות (לא פורמליות) הן בשלבי הפיתוח והן בעת איתור תקלות.
- שימו לב: למנגנוני שפת JAVA, כדוגמת אופרטורים, יש חוזים (לא נשתמש בהגדרות הפורמליות שלהם)

דוגמא: הוכחת נכונות של min1

```
/* precondition:  1) arr != null  2) arr.length > 0
 *                3) arr contains only numbers
 * postcondition: Returns the minimal element in arr    */
  public static double min1(double[] arr) {
        double m = 1.0/0.0; // Infinity
        for (double x: arr)  m = (x < m ? x : m);
        return m;
 }
```

- בהתחלה m==Infinity
- בשל תנאים 2+1, לולאת ה- for תבצע לפחות איטרציה אחת.
- בשל תנאי 3, בסיום האיטרציה הראשונה m==arr[0].
- טענת עזר: לאחר האיטרציה ה- i {m=min{arr[0],..,arr[i-1]}.

הוכחה: באינדוקציה על i. ראינו עבור i=1. נניח נכונות עד i מסויים ונוכיח עבור i+1.
m = min{m,arr[i]} = min{min{arr[0],...,arr[i-1]},arr[i]} = min{arr[0],...,arr[i]}
(השוויון האחרון נובע מתנאי 3).

- מסקנה: לאחר סיום הלולאה m שווה לאיבר המינימלי ב- arr.

min6 :kNdI?

```
/* Same contract as min1 */
  public static double min6(double[] arr) {
          return minInRange(arr, 0, arr.length-1);
  }
 /*   preconditions: 1) arr != null,  2) arr.length > 0,
  *                        3) 0 <= low <= high < arr.length
  *                        4) arr contains only numbers
  * postcondition:
  *          returns the minimal element in {arr[low],...,arr[high]}  */
 public static double minInRange(double[] arr, int low, int high) {
          if ( low==high ) return arr[low];
          int middle = low + (high-low)/2;
          double m1 = minInRange(arr, low,  middle);
          double m2 = minInRange(arr, middle + 1, high  );
          return (m1 < m2 ? m1 : m2);
 }
```

הוכחת נכונות של min6

- min6 מכילה פקודה אחת – קריאה ל- minInRange
- כל הארגומנטים בפקודה ניתנים לחישוב בגלל תנאי 1 של min6
- תנאי הקדם של minInRange מתקיימים:
  - תנאים 1+2+3 של min6 ⇐ תנאים 1+2+4 של minInRange
  - תנאי 2 של min6 ⇐ 0<=arr.length-1 ⇐ תנאי 3 של minInRange
- תנאי האחר של minInRange מבטיח להחזיר את הערך המינימלי ב- {arr[0],...,arr[arr.length-1]}

הוכחת נכונות של minInRange

- הערכים arr[low],...,arr[high] ניתנים לחשוב בשל תנאים 1+3
- המשך ההוכחה הוא באינדוקציה על high-low
- עבור high-low=0: מוחזר arr[low]==min{arr[low],...,arr[high]}
- נניח נכונות עבור high-low=k ונוכיח עבור high-low=k+1>0
- תנאי הקדם מתקיימים עבור שתי הקריאות הרקורסיביות:
  - 0<=low <= middle
  - middle ==low+(high-low)/2 == (high+low)/2 < high
- ניתן להשתמש בהנחת האינדוקציה עבור שתי הקריאות מכיוון:
  - middle-low < high-low
  - high-(middle+1) <= high-low-1 < high-low
- min{arr[low],...,arr[high]} = min{ min{arr[low],...,arr[medium]}, min{arr[medium+1],...,arr[high]}}