

תוכנה 1

סמסטר ב' תשס"ח

תרגול מס' 3
מבני בקרה וחוזים
ליאור שפירא ואוהד ברזילי

Compilation vs. Runtime Errors

שגיאות קומפילציה (הידור): שגיאות שניתן "לתפוס" בעת קריאת הקובץ והפיכתו ל-bytecode ע"י המהדר

דוגמאות:

```
Class MyClass {  
    void f() {  
        int n=10;  
  
    void g() {  
        int m = 20;  
    }  
}
```

```
...  
short x = 5;  
short y = 10;  
short z = x * y;  
...
```

```
...  
int i;  
System.out.println(i);  
...
```

Compilation vs. Runtime Errors

■ שגיאות זמן ריצה: לא ניתן לדעת שתהיה שגיאה במקום ספציפי בזמן ההידור (קומפילציה)

■ דוגמאות:

```
...  
int a[] = new int[10];  
...  
a[15] = 10;  
...
```

```
...  
String s = null;  
System.out.println(s.length());  
...
```

■ מתקשר למנגנון החריגים (exceptions), עליו נלמד בהמשך

Compilation vs. Runtime Errors

האם יש עוד סוג של טעויות? ■

כן, הכי גרועות, טעויות לוגיות בתוכנית ■

```
public class T {
    /** calculate x! */
    public static int factorial(int x) {
        int f = 0;
        for (int i=2;i<=x;i++)
            f = f * I;
        return f;
    }
}
```

If-Else Statement

```
public class Fibonacci {  
    ...  
    /** Returns the n-th Fibonacci element */  
    public static int computeElement(int n) {  
        if (n==0) return 1;  
        else if (n==1) return 1;  
        else return computeElement(n-1) + computeElement(n-2);  
    }  
}
```

Assumption:
 $n \geq 0$

Switch Statement

```
public class Fibonacci {  
    ...  
  
    /** Returns the n-th Fibonacci element */  
    public static int computeElement(int n) {  
        switch(n) {  
            case 0:  
                return 1;  
            case 1:  
                return 1;  
            default:  
                return computeElement(n-1) + computeElement(n-2);  
        }  
    }  
}
```

Assumption:
 $n \geq 0$

Switch Statement

```
public class Fibonacci {  
    ...  
  
    /** Returns the n-th Fibonacci element */  
    public static int computeElement(int n) {  
        switch(n) {  
            case 0:  
                return 1;  
            case 1:  
                return 1;  
                break;  
            default:  
                return computeElement(n-1) + computeElement(n-2);  
        }  
    }  
}
```

Assumption:
 $n \geq 0$

For Loop

- A loop instead of a recursion

```
static int computeElement(int n) {  
    if (n == 0 || n == 1)  
        return 1;  
  
    int prev = 1;  
    int prevPrev = 1;  
    int curr;  
  
    for (int i = 2 ; i < n ; i++) {  
        curr = prev + prevPrev;  
        prevPrev = prev;  
        prev = curr;  
    }  
  
    curr = prev + prevPrev;  
    return curr;  
}
```

Assumption:
 $n \geq 0$

נתונים במקום חישוב

- בתרגום רקורסיה ללולאה אנו משתמשים במשתני עזר לשמירת המצב `prev` ו-`curr` `prevPrev`
- הלולאה "זוכרת" את הנקודה שבה אנו נמצאים בתהליך החישוב
- דין: יעילות לעומת פשטות.
עיקרון ה-KISS (**keep it simple stupid**)
- תרגיל: כתבו את השירות `computeElement` בעזרת `prev` ו-`prevPrev` בלבד (ללא `curr`)

For Loop

- Printing the first n elements:

```
public class Fibonacci {  
    public static int computeElement(int n) {  
        ...  
    }  
  
    public static void main(String[] args) {  
        for(int i = 0 ; i < 10 ; i++)  
            System.out.println(computeElement(i));  
    }  
}
```

מודולריות, שכפול קוד ויעילות

- יש כאן חוסר יעילות מסוים:
- לולאת ה-`for` חוזרת גם ב-`main` וגם ב-`computeElement`. לכאורה, במעבר אחד ניתן גם לחשב את האברים וגם להדפיס אותם
- כמו כן כדי לחשב איבר בסדרה איננו משתמשים בתוצאות שכבר חישבנו (של אברים קודמים) ומתחילים כל חישוב מתחילתו

מודולריות, שכפול קוד ויעילות

- מתודה (פונקציה) צריכה לעשות דבר אחד בדיוק!
- ערוב של חישוב והדפסה פוגע במודולריות (מדוע?)
- היזהרו משכפול קוד!
- קטע קוד דומה המופיע בשתי פונקציות שונות יגרום במוקדם או במאוחר לבאג בתוכנית (מדוע?)
- את בעיית היעילות (הוספת מנגנון memoization) אפשר לפתור בעזרת מערכים (תרגיל)

for vs. while

- The following two statements are almost equivalent:

```
for (int i = 0 ; i < n ; i++)  
    System.out.println (computeElement (i) ) ;
```

```
int i=0;  
while (i < n) {  
    System.out.println (computeElement (i) ) ;  
    i++;  
}
```

while vs. do while

- The following two statements are equivalent if and only if $n > 0$:

```
int i=0;
while (i < n) {
    System.out.println(computeElement(i));
    i++;
}
```

```
int i=0;
do {
    System.out.println(computeElement(i));
    i++;
} while (i < n);
```



חוזים



חוזה בין ספק ללקוח

- 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 למצב 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 !?

דוגמא 1

```
/*
 * 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
יקוימו כל התנאים בתנאי הקדם?
?arr==null
?arr.length == 0
?NaN מכיל arr
?–Infinity או Infinity מכיל arr

דוגמא 2 (אותו קוד, חוזה 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;
}
```

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;
}
```

(precondition אָלל) 4 אָמאָד

```
/*
 * precondition: true
 *
 * 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 אָלל) 5 אָמגוּד

```
/*
 * precondition: true
 *
 * 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;
}
```

הוכחת נכונות של שירותים

■ 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 1+2, לולאת ה-`for` תבצע לפחות איטרציה אחת.

■ בשל תנאי 3, בסיום האיטרציה הראשונה `m==arr[0]`.

■ טענת עזר: לאחר האיטרציה ה-`i` $m = \min\{arr[0], \dots, arr[i-1]\}$

■ הוכחה: באינדוקציה על `i`. ראינו עבור `i=1`. נניח נכונות עד `i` מסויים ונוכיח עבור

$$m = \min\{m, arr[i]\} = \min\{\min\{arr[0], \dots, arr[i-1]\}, arr[i]\} = \min\{arr[0], \dots, arr[i]\}$$

(השוויון האחרון נובע מתנאי 3).

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

min6 : אמת

```
/* 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` \Leftarrow תנאים $1+2+4$ של `minInRange`
- תנאי 2 של `min6` $\Leftarrow 0 \leq arr.length-1 \Leftarrow$ תנאי 3 של `minInRange`
- תנאי האחר של `minInRange` מבטיח להחזיר את הערך המינימלי ב- $\{arr[0], \dots, arr[arr.length-1]\}$

הוכחת נכונות של `minInRange`

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