

תוכנה 1 בשפת Java

תרגול מספר 3 - מבני בקרה

אורנית דרור ואוהד ברזילי
13,14 בנובמבר 2005

מוסכמות קוד

- חשיבות:
 - 80% מעלות פיתוח תוכנה הוא עבור תחזוקת קוד
 - כמעט ואין קוד שמתוחזק רק ע"י הכותב המקורי שלו
 - מוסכמות משפרות את קריאות הקוד ומאפשרות להבינו מהר
- קריאות של קוד היא עניין סובייקטיבי והסגנונות מרובים
- מוסכמות Sun: (24 pages) <http://java.sun.com/docs/codeconv>
- דוגמא: שמות מזהים ארוכים:
 - המילים יופרדו עם קו תחתי ("is_odd")
 - שיטת הגמל: אות גדולה בתחילת מילה ("isOdd")
- חשוב לשמור על אחידות!

מוסכמת
Sun

אינדנטציה

- אינדנטציה (Indentation) = עימוד קוד, הזחה
- מסולסליים בסוף השורה או בשורה נפרדת

```
if (condition)  
{  
    statements;  
}
```

```
if (condition) {  
    statements;  
}
```

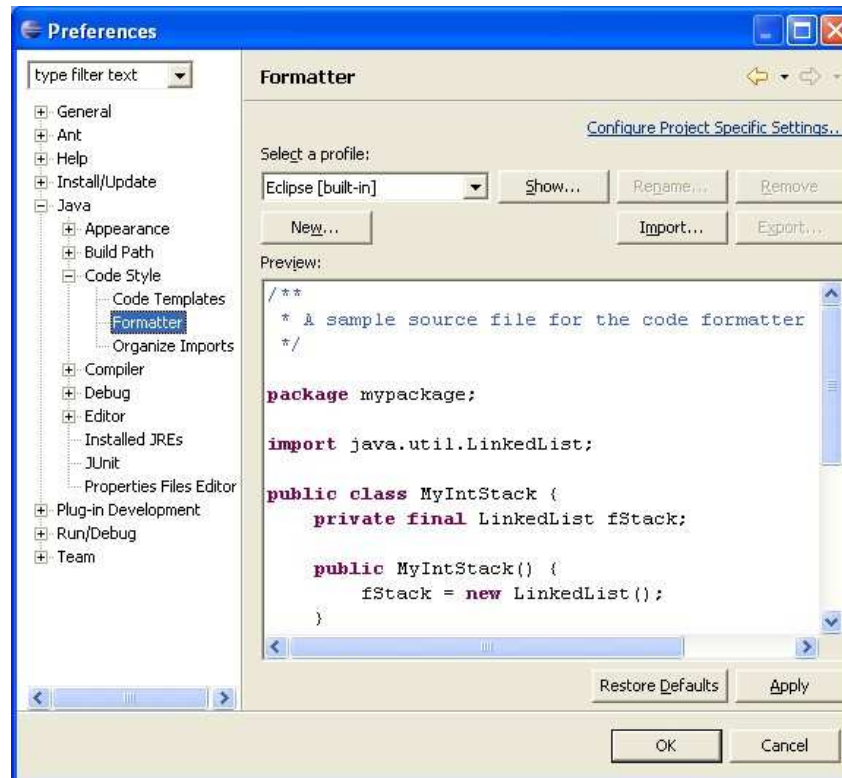
מוסכמת
Sun

- בלוק של שורה ייעטף במסולסליים למרות שמבחינת השפה אין צורך בהן
- הערה: בשקפים כדי לחסוך במקום נוותר על דרישה זו.

מוסכמת C

אינדנטציה

- ניתן 'ללמד' סביבת פיתוח מודרנית מהו הסגנון הרצוי
- באקליפס: Window->preferences->java->code style->formatter



מרקורסיה ללולאה

- בשפת Scheme פונקציה היא הרכבה של פעולות (של פונקציות אחרות) - **תכנות פונקציונאלי**
- בשפת Java ניתן לראות פונקציה גם כסדרה של פעולות – **תכנות פרוצדוראלי או אימפרטיבי**
- נדגים איך ניתן לתרגם את הרקורסיה לחישוב סדרת פיבונאצ'י ללולאה

סדרת פיבונאצ'י

□ תזכורת:

1, 1, 2, 3, 5, 8, 13, 21, 34,...

$$a_0 = 1$$

$$a_1 = 1$$

$$a_n = a_{n-1} + a_{n-2}$$

□ מוטיבציה: ארנבים אידיאליים

סדרת פיבונאצ'י

```
package il.ac.tau.cs.software1.recitation3;

public class Fibonacci {
    ...

    /**
     * Returns the n-th Fibonacci element
     * @pre n >= 0 , "n is non-negative"
     */
    static int computeElement(int n) {
        if (n == 0)
            return 1;
        if (n == 1)
            return 1;
        return computeElement(n-1)+computeElement(n-2);
    }
}
```

אין צורך ב-else
בגלל ה- return

הערות

- נשים לב שחבילת העבודה (**package**) נכתבת מהכלל את הפרט (שם ה-domain). נדון בזה בהמשך הקורס.
- המלל שבהערה `/** ... */` יכלל בתיעוד האוטומטי של המחלקה. פרטים בהמשך הקורס
- התגית **@pre** היא סימון מקובל לתנאי קדם (סימון מקובל אחר: **@require**)
- המילה השמורה **static**, משמעה שהשרות (מתודה, פונקציה) לא משויך לעצם מסוים. נדון על כך בהמשך.
- המילה **int** מופיעה פעמיים בהגדרת הפונקציה:
 - כדי לציין את טיפוס הארגומנט
 - כדי לציין את טיפוס הערך המוחזר

שימוש ב- else ... if

```
package il.ac.tau.cs.software1.recitation3;

public class Fibonacci {
    ...

    /**
     * Returns the n-th Fibonacci element
     * @pre n >= 0 , "n is non-negative"
     */
    static int computeElement(int n) {
        if (n == 0) {
            return 1;
        } else {
            if (n == 1) {
                return 1;
            } else {
                return computeElement(n-1)+computeElement(n-2);
            }
        }
    }
}
```

קיצור על ידי else if

```
package il.ac.tau.cs.software1.recitation3;

public class Fibonacci {
    ...
    /**
     * Returns the n-th Fibonacci element
     * @pre n >= 0 , "n is non-negative"
     */
    static int computeElement(int n) {
        if (n == 0) {
            return 1;
        } else if (n == 1) {
            return 1;
        } else {
            return computeElement(n-1)+computeElement(n-2);
        }
    }
}
```

else if מתאים כאשר
ממספר אפשרויות תתקיים
תמיד אפשרות אחת בדיוק

האופרטור ||

```
package il.ac.tau.cs.software1.recitation3;

public class Fibonacci {
    ...

    /**
     * Returns the n-th Fibonacci element
     * @pre n >= 0 , "n is non-negative"
     */
    static int computeElement(int n) {
        if ((n == 0) || (n == 1)) {
            return 1;
        } else {
            return computeElement(n-1)+computeElement(n-2);
        }
    }
}
```

קדימות האופרטור ||

- מתכנתים ותיקים (בעיקר מתכנתי C) יטענו שהביטוי $(n == 0) || (n == 1)$ מסורבל ומגושם
- קדימות האופרטורים היא כזו שהשימושים השכיחים יכילו מעט זוגות סוגריים, כלומר: הביטוי לעיל שקול לביטוי: $(n == 0) || (n == 1)$
- במקרה זה, יטענו הותיקים, שתי זוגות הסוגריים אינם הופכים את הקוד לקריא יותר אלא לקריא פחות

וארציות על הנושא: switch

```
package il.ac.tau.cs.software1.recitation3;

public class Fibonacci {
    ...

    /**
     * Returns the n-th Fibonacci element
     * @pre n >= 0 , "n is non-negative"
     */
    static int computeElement(int n) {
        switch(n) {
            case 0:
                return 1;
            case 1:
                return 1;
            default:
                return computeElement(n-1) + computeElement(n-2);
        }
    }
}
```

הערות

- משפטי ה **case** מוזחים
- למה אין צורך ב- **break**?
- מתי נשתמש ב- **switch** ומתי ב- **else if**?
- ניתן לוותר על משפט ה-**default** ולהוציא את הקריאה הרקורסיבית מחוץ ל-**switch**
- הקוד בשקף הבא שקול. מה ניתן לומר על משפט ה-**break**?

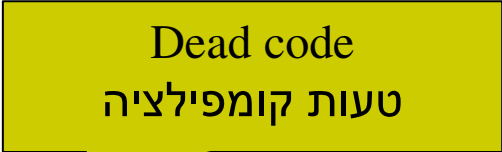
וארציות על הנושא: switch

```
package il.ac.tau.cs.software1.recitaton3;

public class Fibonacci {

    ...

    /**
     * Returns the n-th Fibonacci element
     * @pre n >= 0 , "n is non-negative"
     */
    static int computeElement(int n) {
        switch(n) {
            case 0:
            case 1:
                return 1;
                break;
            default:
                return computeElement (n-1)+computeElement (n-2);
        }
    }
}
```



Dead code
טעות קומפילציה

הדפסת n האברים הראשונים

□ נרצה להשתמש בשרות `computeElement` כדי להדפיס את n האברים הראשונים בסדרת פיבונאצ'י

□ נכתוב את השרות הסטטי `printNElements` אשר מקבל כארגומנט את מספר האברים הרצוי וקורא בלולאה לשרות הסטטי `computeElement`

הדפסת n האברים הראשונים

```
package il.ac.tau.cs.software1.recitation3;

public class Fibonacci {
    public static void main(String[] args) {
        printNElements(10);
    }

    /**
     * Prints the first n elements in Fibonacci series
     * @pre n >= 1 , "n is positive"
     */
    static void printNElements(int n) {
        for(int i = 0; i < n; i++)
            System.out.println(computeElement(i));
    }
    ...
    static int computeElement(int n) {...}
}
```

רצוי לקבל
את n כקלט

הערות

- המשתנה `i` מוגדר בתוך גוף ה-`for`
- אתחול `i` לאפס וההשוואה האם הוא קטן ממש ממספר הפעמים הרצוי היא "קלישאת תכנות"
- הפונקציה `main` כמעט ואינה מכילה דבר
- זוהי נקודת הכניסה (entry point) לתוכנית, ומשיקולי מודולריות רצוי להשאיר אותה "רזה"

הערות

- נחזור לשרות `computeElement`
- נרצה להחליף את הרקורסיה בלולאה
- נממש מחדש את השירות כך שידמה את התהליך שבו אנו מחשבים את הסדרה מהתחלה לסוף
- הערה: שם השרות לא משתנה כי הוא לא צריך להעיד על המימוש

פיבונאצ'י בלולאה

```
/**
 * Returns the n-th Fibonacci element
 */
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;
}
```

נתונים תמורת חישוב

- בתרגום רקורסיה ללולאה אנו משתמשים במשתני עזר לשמירת המצב `prev`, `curr`, ו-`prevPrev`
- במובן מסוים אנו רוצים שהלולאה "תזכור" את הנקודה שבה אנו נמצאים בתהליך החישוב
- תרגיל: כתבו את השירות `computeElement` תוך שימוש ב-`prev` וב-`prevPrev` בלבד (ללא `curr`)
 - דיון: יעילות לעומת פשטות (KISS principle)

מודולריות, שכפול קוד ויעילות

□ הדפסת n האברים בסדרה ע"י לולאה הקוראת ל-`computeElement`:

```
static void printNElements(int n) {  
    for(int i=0; i<n; i++)  
        System.out.println(computeElement(i));  
}
```

□ יש כאן חוסר יעילות מסוים:

■ לולאת ה-`for` חוזרת גם בשירות `printNElements` וגם בשירות `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 (היזכרו בדוגמת memoization מקורס scheme)

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?)
 - נחזור למימוש האיטרטיבי ונמצא את שכפול הקוד. על מה הדבר מעיד? איך נתקן אותו?
- את בעיית היעילות (הוספת מנגנון memoization) נפתור בקלות אחרי שנלמד מערכים

for vs. while

- לולאות for ו-while בשפת Java הן שקולות (סוכר תחבירי)
- נתרגם את printNElements ללולאת while:

```
static void printNElements(int n) {  
    for(int i = 0 ; i < n ; i++)  
        System.out.println(computeElement(i));  
}
```

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


while vs. do while

```
/** @pre: n >= 1 , "n is positive" */
static void printNElements(int n) {
    int i = 0;
    while(i < n) {
        System.out.println(computeElement(i));
        i++;
    }
}
```

```
/** @pre: n >= 1 , "n is positive" */
static void printNElements(int n) {
    int i = 0;
    do {
        System.out.println(computeElement(i));
        i++;
    } while (i < n);
}
```

עובד בגלל תנאי
הקדם $n > 1$

שאלות?
