
תוכנה 1 בשפת Java

שיעור מספר 3: "זו זכות לתת שרות"

מקוצר: סמסטר ב' 2007

בית הספר למדעי המחשב
אוניברסיטת תל אביב



עקרונות תכנות נכונים



"משבר התוכנה"

- כתיבת תוכנית קטנה אינה משימה קשה

- הקושי בהנדסת תוכנה נעוץ בבניית מערכות תוכנה גדולות ($>1,000,000$ שורות קוד), ע"י מספר אנשים הדורשות תחזוקה לאורך זמן רב (כמה שנים)

- הגודל כן קובע

- מאז סוף שנות ה-60 של המאה ה-20 ברור לעולם התוכנה, כי בשביל להתגבר על הקושי שבכתיבת מערכות גדולות יש צורך להלביש על שפת התכנות עקרונות פיתוח נכונים אשר ישפרו את היכולת לכתוב מערכות תוכנה מורכבות

תכנות מונחה עצמים

■ אחד מאוספי העקרונות האלה מכונה "תכנות מונחה עצמים" (Object Oriented Programming) שעיקריו (כפי שיבואו לידי ביטוי בקורס):

■ שימוש חוזר בקוד

■ הפשטה

■ מודולריות

■ תיכון בעזרת חוזים (design by contract)

■ ביצוע מקסימום בדיקות תקינות בזמן קומפילציה

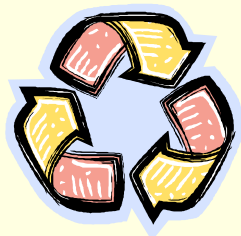
■ ניהול זיכרון אוטומטי

■ חלק משפות התכנות המודרניות מקדמות עקרונות אלו ע"י הגדרת מבנים בשפה שיתמכו בהם בצורה ישירה או עקיפה

■ שפת Java היא שפה כזו

שימוש חוזר בתוכנה

- על מנת לשמור על עלויות תוכנה סבירות, יש לשפר את תפוקת מפתחי התוכנה
- שיפור תפוקה יומית של מתכנת דורש שיפורים משמעותיים בתהליכי הפיתוח, שפות התכנות, וכלי הפיתוח
- בנוסף, ניתן להקטין את עלות הפיתוח ע"י שימוש ברכיבי תוכנה קיימים, שפותחו עבור פרויקט קודם או פותחו במיוחד כתשתית לארגון
- שימוש חוזר בתוכנה כרוך בקשיים רבים, לא כולם טכניים: תסמונת "לא הומצא אצלנו", תשלום עבור תוכנה לפי שורות קוד
- הניסיון מראה שרכיבי תוכנה מונחת עצמים מתאימים לשימוש חוזר יותר מרכיבים פרוצדורלים



מודולריות

- מודולריות היא תכונה חשובה של תוכנה
- נחוצה כדי לאפשר הפרדת עניינים בזמן הפיתוח, ולשפר קריאות לצורך תחזוקה
- מודולריות פירושה היכולת לפרק מערכת למרכיבים, לבנות מערכת ממרכיבים, להבין כל מודול בפני עצמו
- מודולריות טובה כתכונה של מערכת דורשת מודולים בעלי חוזק פנימי גבוה, וצמידות נמוכה
- מתברר שארכיטקטורת מערכת שמבוססת על הנתונים מאפשרת מודולריות טובה יותר מארכיטקטורה שמבוססת על הפונקציונליות
- מכאן היתרון של פיתוח תוכנה מונחה עצמים

Java כשפה מונחית עצמים

- לאחר שכיסינו את רוב היסודות הפרוצדורלים של שפת Java, נקדיש את רוב הקורס למבנים בשפה אשר מקדמים עקרונות תכנות נכונים
- ננסה להבין עבור כל מבנה תחבירי כזה – איך השימוש בו ישפר את איכות הקוד הנוצר



שרותים

- לשימוש בשרותים יש מרכיב מרכזי בבניית מערכות תוכנה גדולות בכמה מישורים:
 - חסכון בשכפול קוד
 - עליה ברמת ההפשטה
 - הגדרת יחסי ספק-לקוח בין כותב השרות והמשתמשים בשרות



שרותים - חסכון בשכפול קוד

- אם קטע קוד מופיע יותר מפעם אחת (copy-paste) יש להפוך אותו לפונקציה (שרות)
- אם הקוד המופיע דומה אבל לא זהה יש לבדוק האם אפשר לבטא את השוני כפרמטר לשרות

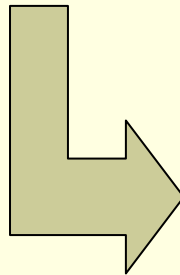
שרותים והפשטה

- גם אם אין חסכון בשכפול קוד יש חשיבות בהפיכת קוד למתודה
- המתודה מתפקדת כקופסא שחורה המאפשרת לקורא הקוד להבין את הלוגיקה שלו בקלות, ולתחזק אותו ביעילות
 - "מרב עצים לא רואים את היער"
- שיקולי יעילות (קפיצה נוספת למתודה מאיטה במעט את ריצת הקוד) הם משניים בשיקולי פיתוח מערכות תוכנה גדולות
 - קומפילרים חכמים, אופטימיזצורים ומעבדים חזקים משמעותיים בהרבה

שרותים והפשטה

דוגמא

```
public static void printOwing(double amount) {  
    //printBanner  
    System.out.println("*****");  
    System.out.println("*** Customer Owes ***");  
    System.out.println("*****");  
  
    //print details  
    System.out.println ("name:" + name);  
    System.out.println ("amount" + amount);  
}
```



```
public static void printOwing(double amount) {  
    printBanner();  
    printDetails(amount);  
}  
  
public static void printBanner() {  
    System.out.println("*****");  
    System.out.println("*** Customer Owes ***");  
    System.out.println("*****");  
}  
  
public static void printDetails(double amount) {  
    System.out.println ("name:" + name);  
    System.out.println ("amount" + amount);  
}
```



שכתוב מבני (refactoring)

- ישנן פעולות של שכתוב קוד שהן כל כך שכיחות עד שהומצא להן שם לדוגמא: הפיכת קטע קוד לשרות שראינו בשקף הקודם נקרא: "חלץ למתודה" (extract method)
- בשנים האחרונות נאסף מספר גדול של פעולות כאלה וקובץ בקטלוג בשם Refactoring. הקטלוג זמין ברשת ובכמה ספרים
<http://www.refactoring.com/catalog/index.html>
- סביבות פיתוח מודרניות (לרבות Eclipse) מאפשרות שכתובים אוטומטיים בלחיצת כפתור
- ביצוע שכתוב בעזרת כלי אוטומטי פותר בעיות רבות של חוסר עקביות העשויות להיווצר כאשר הוא מתבצע ידנית
- למשל: החלפת שם משתנה בצורה עקבית או חילוץ למתודה קטע קוד התלוי במשתנה מקומי

לקוח וספק במערכת תוכנה

- **ספק** (supplier) – הוא מי שקוראים לו (לפעמים נקרא גם שרת, server)
- **לקוח** (client) הוא מי שקרא לספק או מי שמשתמש בו (לפעמים נקרא גם משתמש, user). דוגמא:

```
public static void do_something() {  
    // doing...  
}
```

```
public static void main(String [] args) {  
    do_something();  
}
```

- בדוגמא זו הפונקציה `main` היא **לקוחה** של הפונקציה `do_something()`
- `do_something` היא **ספקית** של `main`

לקוח וספק במערכת תוכנה

- הספק והלקוח עשויים להיכתב בזמנים שונים, במקומות שונים וע"י אנשים שונים ואז כמובן לא יופיעו באותו קובץ (באותה מחלקה)

```
public static void do_something() {  
    // doing...  
}
```

Supplier.java

```
public static void main(String [] args) {  
    do_something();  
}
```

Client.java



- חלק נכבד בתעשיית התוכנה עוסק בכתיבת **ספריות** – מחלקות המכילות אוסף שרותים שימושיים בנושא מסוים
- כתב הספרייה נתפס כספק שרותים בתחום (domain) מסוים

המחלקה כספריה של שרותים

■ ניתן לראות במחלקה ספריה של שרותים: אוסף של פונקציות עם מכנה משותף

■ רוב המחלקות ב Java, נוסף על היותן ספריה, משמשות גם כטיפוס נתונים. ככאלו הן מכילות רכיבים נוספים פרט לשרותי מחלקה. נדון במחלקות אלו בהמשך הקורס

■ ואולם קיימות ב- Java גם כמה מחלקות המשמשות כספריות בלבד. בין השימושיות שבהן:

- `java.lang.Math`
- `java.util.Arrays`
- `java.lang.System`

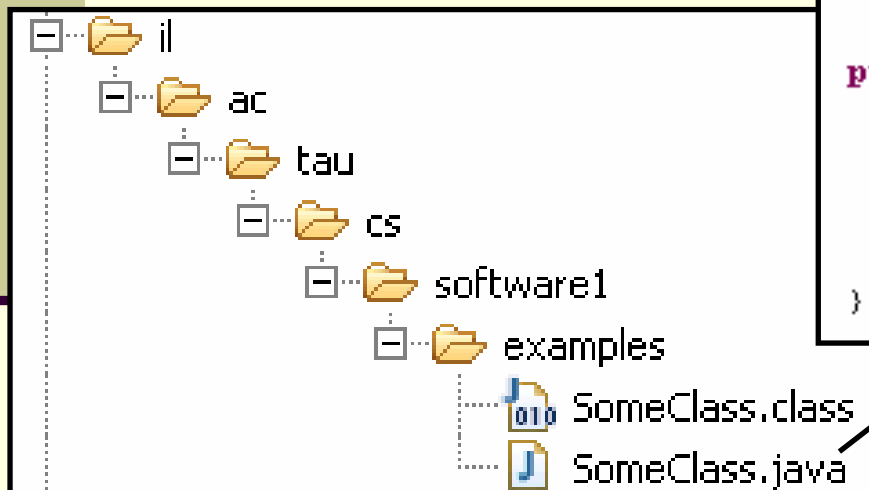


חבילות ומרחב השמות

- מרחב השמות של Java היררכי
 - בדומה לשמות תחומים באינטרנט או שמות תיקיות במערכת הקבצים
- חבילה (package) יכולה להכיל מחלקות או תת-חבילות בצורה רקורסיבית
- שמה המלא של מחלקה (fully qualified name) כולל את שמות כל החבילות שהיא נמצאת בהן מהחיצונית ביותר עד לפנימית. שמות החבילות מופרדים בנקודות
- מקובל כי תוכנה הנכתבת בארגון מסוים משתמש בשם התחום האינטרנטי של אותו ארגון כשם החבילות העוטפות

חבילות ומרחב השמות

- קיימת התאמה בין מבנה התיקיות (directories, folders) בפרויקט תוכנה ובין חבילות הקוד (packages)



```
package il.ac.tau.cs.software1.examples;  
  
public class SomeClass {  
  
    public static void main(String[] args) {  
        //...  
    }  
}
```

משפט import

שימוש בשמה המלא של מחלקה מסרבל את הקוד:

```
System.out.println("Before: x=" +  
java.util.Arrays.toString(arr));
```

ניתן לחסוך שימוש בשם מלא ע"י ייבוא השם בראש הקובץ (מעל הגדרת המחלקה)

```
import java.util.Arrays;  
...  
System.out.println("Before: x=" + Arrays.toString(arr));
```

משפט import

■ כאשר עושים שימוש נרחב במחלקות מחבילה מסויימת ניתן לייבא את שמות כל המחלקות במשפט import יחיד:

```
import java.util.*;
```

```
...
```

```
System.out.println("Before: x=" + Arrays.toString(arr));
```

■ השימוש ב-* אינו רקורסיבי, כלומר יש צורך במשפט import נפרד עבור כל תת חבילה:

```
// for classes directly under subpackage
```

```
import package.subpackage.*;
```

```
// for classes directly under subsubpackage1
```

```
import package.subpackage.subsubpackage1.*;
```

```
// only for the class someClass
```

```
import package.subpackage.subsubpackage2.someClass;
```

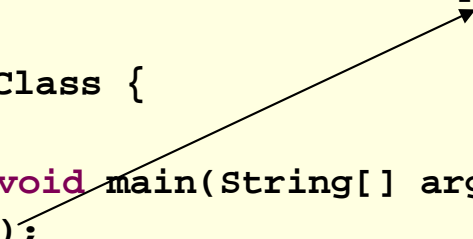
משפט static import

- החל מ Java5 ניתן לייבא למרחב השמות את השרות או המשתנה הסטטי (static import) ובכך להימנע מציון שם המחלקה בגוף הקוד:

```
package il.ac.tau.cs.software1.examples;
import static il.ac.tau.cs.software1.examples.SomeOtherClass.someMethod;

public class SomeClass {

    public static void main(String[] args) {
        someMethod();
    }
}
```



- גם ב static import ניתן להשתמש ב- *



הערות על מרחב השמות ב- Java

- שימוש במשפט `import` אינו שותל קוד במחלקה והוא נועד לצורכי נוחות בלבד
- אין צורך לייבא מחלקות מאותה חבילה
- אין צורך לייבא את החבילה `java.lang`
- ייבוא כוללני מדי של שמות מעיד על צימוד חזק בין מודולים
- ייבוא של חבילות עם מחלקות באותו שם יוצר `ambiguity` של הקומפיילר וגורר טעות קומפילציה ("התנגשות שמות")



CLASSPATH

- איפה נמצאות המחלקות?
- איך יודעים הקומפילר וה-JVM היכן לחפש את המחלקות המופיעות בקוד המקוד או ה-byte code?
- קיים משתנה סביבה בשם **CLASSPATH** המכיל שמות של תיקיות במערכת הקבצים שם יש לחפש מחלקות הנזכרות בתוכנית
- ה-**CLASSPATH** מכיל את תיקיות ה"שורש" של חבילות המחלקות ניתן להגדיר את המשתנה בכמה דרכים:
 - הגדרת המשתנה בסביבה (תלוי במערכת ההפעלה)
 - הגדרה אד-הוק – ע"י הוספת תיקיות חיפוש בשורת הפקודה (בעזרת הדגל cp או classpath)
 - הגדרת תיקיות החיפוש בסביבת הפיתוח

jar

- כאשר ספקי תוכנה נותנים ללקוחותיהם מספר גדול של מחלקות הם יכולים לארוז אותן כארכיב
- התוכנית **jar** (Java **AR**chive) אורזת מספר מחלקות לקובץ אחד תוך שמירה על מבנה החבילות הפנימי שלהן
- הפורמט תואם למקובל בתוכנות דומות כגון zip, tar, rar ואחרות
- כדי להשתמש במחלקות הארוזות אין צורך לפרוס את קובץ ה-**jar**
 - ניתן להוסיפו ל `CLASSPATH` של התוכנית
- התוכנית **jar** היא חלק מה- JDK וניתן להשתמש בה משורת הפקודה או מתוך סביבת הפיתוח



API and javadoc

- קובץ ה- jar עשוי שלא להכיל קובצי מקור כלל, אלא רק קובצי class (למשל משיקולי זכויות יוצרים)
- איך יכיר לקוח שקיבל jar מספק תוכנה כלשהו את הפונקציות והמשתנים הנמצאים בתוך ה- jar, כדי שיוכל לעבוד איתו?
- בעולם התוכנה מקובל לספק ביחד עם הספריות גם מסמך תיעוד, המפרט את שמות וחתימות את המחלקות, השרותים והמשתנים יחד עם תיאור מילולי של אופן השימוש בהם
- תוכנה בשם javadoc מחוללת **תיעוד אוטומטי** בפורמט html על בסיס הערות התיעוד שהופיעו בגוף קובצי המקור
- תיעוד זה מכונה API (Application Programming Interface)
- תוכנת ה javadoc היא חלק מה- JDK וניתן להשתמש בה משורת הפקודה או מתוך סביבת הפיתוח


```
/** Documetntaion for the package */  
package somePackage;
```

```
/** Documetntaion for the class  
 * @author your name here  
 */
```

```
public class SomeClass {
```

```
/** Documetntaion for the class variable */  
public static int someVariable;
```

```
/** Documetntaion for the class method  
 * @param x documentation for parameter x  
 * @param y documentation for parameter y  
 * @return  
 *     documentation for return value  
 */
```

```
public static int someMethod(int x, int y, int z){  
    // this comment would NOT be included in the documentation  
    return 0;  
}  
}
```

Java API

■ חברת Sun תיעדה את כל מחלקות הספרייה של שפת Java וחוללה עבורן בעזרת javadoc אתר תיעוד מקיף ומלא הנמצא ברשת:

<http://java.sun.com/j2se/1.5.0/docs/api/>

תיעוד וקוד

- בעזרת מחולל קוד אוטומטי הופך התיעוד לחלק בלתי נפרד מקוד התוכנית
- הדבר משפר את הסיכוי ששינויים עתידיים בקוד יופיעו מיידית גם בתיעוד וכך תשמר העקביות בין השניים



פערי הבנה

- חתימה אינה מספיקה, מכיוון שהספק והלקוח אינם רק שני רכיבי תוכנה נפרדים אלא גם לפעמים נכתבים ע"י מתכנתים שונים עשויים להיות פערי הבנה לגבי תפקוד שרות מסוים

- הפערים נובעים ממגבלות השפה הטבעית, פערי תרבות, הבדלי אינטואיציות, ידע מוקדם ומקושי יסודי של תיאור מלא ושיטתי של עולם הבעיה

- לדוגמא: נתבונן בשרות `divide` המקבל שני מספרים ומחזיר את המנה שלהם:

```
public static int divide(int numerator, int denominator)
{...}
```

- לרוב הקוראים יש מושג כללי נכון לגבי הפונקציה ופעולתה
- למשל, די ברור מה תחזיר הפונקציה אם נקרא לה עם הארגומנטים 6 ו-2

"Let us speak of the unspeakable"

- אך מה יוחזר עבור הארגומנטים 7 ו-2 ?
 - האם הפונקציה מעגלת למעלה?
 - מעגלת למטה?
 - ועבור ערכים שליליים?
 - אולי היא מעגלת לפי השלם הקרוב?

- ואולי השימוש בפונקציה **אסור** בעבור מספרים שאינם מתחלקים ללא שארית?



- מה יקרה אם המכנה הוא אפס?
 - האם נקבל ערך מיוחד השקול לאינסוף?
 - האם קיים הבדל בין אינסוף ומינוס אינסוף?

- ואולי השימוש בפונקציה **אסור** כאשר המכנה הוא אפס?

- מה קורה בעקבות שימוש **אסור** בפונקציה?
 - האם התוכנית **תעוף**?
 - האם מוחזר **ערך שגיאה**? אם כן, איזה?
 - האם קיים משתנה או מנגנון שבאמצעותו ניתן לעקוב אחרי שגיאות שארעו בתוכנית?

יותר מדי קצוות פתוחים...

- אין בהכרח תשובה נכונה לגבי השאלות על הצורה שבה על divide לפעול
- ואולם יש לציין במפורש:
 - מה היו ההנחות שביצע כותב הפונקציה
 - במקרה זה הנחות על הארגומנטים (האם הם מתחלקים, אפס במכנה וכו')
 - מהי התנהגות הפונקציה במקרים השונים
 - בהתאם לכל המקרים שנכללו בהנחות
- פרוט ההנחות וההתנהגויות השונות מכונה החוזה של הפונקציה
- ממש כשם שבעולם העסקים נחתמים חוזים בין ספקים ולקוחות
 - קבלן ודיירים, מוכר וקונים, מלון ואורחים וכו'...



עיצוב על פי חוזה (design by contract)

- בשפת Java אין תחביר מיוחד כחלק מהשפה לציון החוזה, ואולם אנחנו נתבסס על תחביר המקובל במספר כלי תכנות
- נציין בהערות התיעוד שמעל כל פונקציה:
 - **תנאי קדם (precondition)** – מהן **ההנחות** של כותב הפונקציה לגבי הדרך התקינה להשתמש בה
 - **תנאי בתר (תנאי אחר, postcondition)** – **מה עושה הפונקציה**, בכל אחד מהשימושים התקינים שלה
- נשתדל לתאר את תנאי הקדם ותנאי הבתר במונחים של ביטויים בולאנים חוקיים ככל שניתן (לא תמיד ניתן)
- שימוש בביטויים בולאנים חוקיים:
 - מדויק יותר
 - יאפשר לנו בעתיד לאכוף את החוזה בעזרת כלי חיצוני

