תוכנה 1 בשפת Java

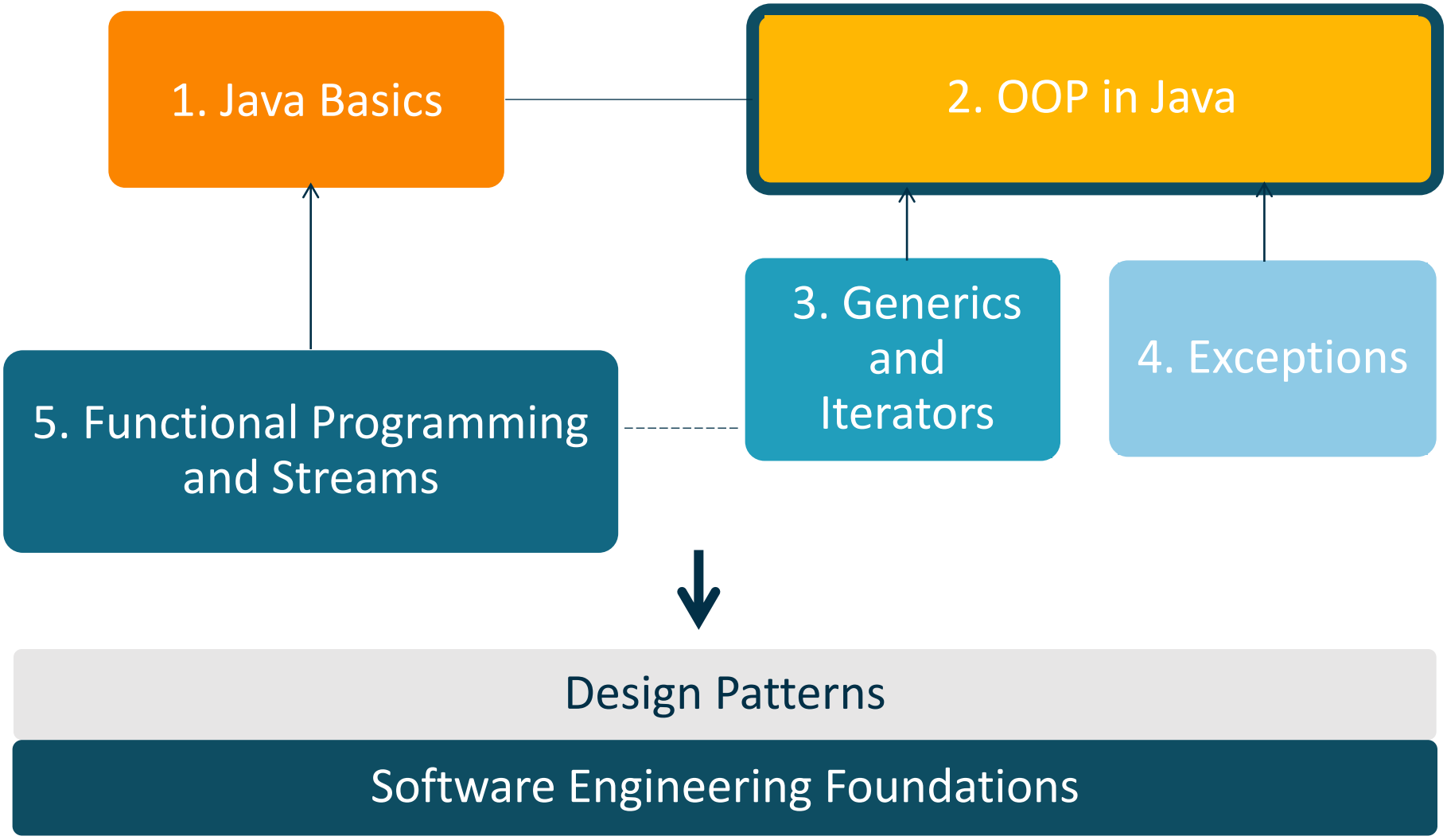
שיעור מספר 3 (חלק שני): חוזים

מיכל קליינבורט

בית הספר למדעי המחשב

אוניברסיטת תל אביב

נושאי הקורס


1. Java Basics
2. OOP in Java
3. Generics and Iterators
4. Exceptions
5. Functional Programming and Streams
Design Patterns
Software Engineering Foundations

התוכנית להיום

חזרה על רצף המספרים – סיבוב

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)

דוגמה:

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

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

- בדוגמה זו הפונקציה **main** היא **לקוחה** של הפונקציה **doSomething()**
- **doSomething** היא **ספקית** של **main**

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)

דוגמה:

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

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

- בדוגמה זו הפונקציה **main** היא **לקוחה** של הפונקציה **doSomething()**
- **doSomething** 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 doSomething(){
  // doing...
}
```

Supplier.java

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

Client.java

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

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

- לדוגמה: נתבונן בשרות **divide** המקבל שני מספרים ומחזיר את המנה שלהם:

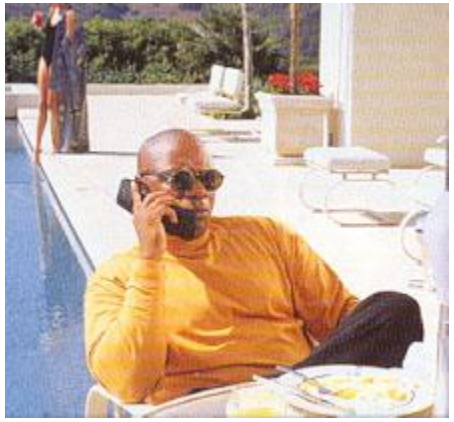
```
public static int divide(int numerator, int denominator)
{...}
```

  - 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

חוזה אפשרי ל- divide

$$\$ret * denominator + (numerator\%denominator) == numerator$$

$\$ret$: $\left\lfloor \frac{numerator}{denominator} \right\rfloor$

$(numerator\%denominator)$: השארית של $\frac{numerator}{denominator}$

$numerator$: מונה

$$|\$ret * denominator| \leq |numerator| \Rightarrow |\$ret| \leq |\frac{numerator}{denominator}|$$

```
public static int divide(int numerator, int denominator)
```

- התחביר מבוסס על כלי בשם Jose
- לפעמים החוזה ארוך יותר מגוף הפונקציה
- ישנים חוזים אפשריים אחרים לפונקציה divide

שרות לעולם לא יבדוק את תנאי הקדם שלו

- שרות לעולם לא יבדוק את תנאי הקדם שלו
- גם לא "ליתר ביטחון"
  - אם שרות בודק תנאי קדם ופועל לפי תוצאת הבדיקה, אזי יש לו התנהגות מוגדרת היטב עבור אותו תנאי – כלומר הוא אינו תנאי קדם עוד

- אי הבדיקה מאפשרת כתיבת מודולים "סובלניים" שיעטפו קריאות למודולים שאינם מניחים דבר על הקלט שלהם
- כך נפריד את בדיקות התקינות מהלוגיקה העסקית (business logic) כלומר ממה שהפונקציה עושה באמת
- גישת תיכון ע"פ חוזה סותרת גישה בשם "תכנות מתגונן" (defensive programming) שעיקריה לבדוק תמיד הכל

חלוקת אחריות

- אבל מה אם הלקוח שכח לבדוק?
- זו הבעיה שלו!

- החוזה מגדיר במדויק אחריות ואשמה, זכויות וחובות:
  - הלקוח – חייב למלא אחר תנאי הקדם לפני הקריאה לפונקציה (אחרת הספק לא מחויב לדבר)
  - הספק – מתחייב למילוי כל תנאי האחר אם תנאי הקדם התקיים

- הצד השני של המטבע – לאחר קריאה לשרות אין צורך לבדוק שהשרות בוצע.
- ואם הוא לא בוצע? יש לנו את מי להאשים...

דוגמה - חיפוש בינארי

```
/**
 * @param a Am array sorted in ascending order
 * @param x a number to be searched in a
 * @return the first occurrence of x in a, or -1 if not
 *         exists
 *
 * @pre "a is sorted in ascending order"
 */
public static int findInSortedArray(int [] a, int x)
```

- האם עליה לבדוק את תנאי הקדם?
- כמובן שלא, בדיקה זו עשויה להיות איטית יותר מאשר ביצוע החיפוש עצמו
- ונניח שהיתה בודקת, מה היה עליה לעשות במקרה שהמערך אינו ממוין?
  - להחזיר 1- ?
  - למיין את המערך?
  - לחפש במערך הלא ממוין?
- על `findInSortedArray` לא לבדוק את תנאי הקדם. אם לקוח יפר אותו היא עלולה להחזיר ערך שגוי או אפילו לא להסתיים אבל זו כבר לא אשמתה...

נכונות התוכנה שכתבנו

- האם התוכנה שכתבנו **נכונה**?
- איך נגדיר **נכונות**?
- **משתמר** (שמורה, invariant) – הוא ביטוי בולאני שערכו נכון 'תמיד'
- נוכיח כי התוכנה שלנו נכונה ע"י כך שנגדיר עבורה משתמר, **ונוכיח** שערכו true בכל רגע נתון
- להוכחה פורמלית (בעזרת לוגיקה) יש חשיבות מכיוון שהיא מנטרלת את **דו המשמעיות** של השפה הטבעית וכן היא לא מניחה דבר על **אופן השימוש** בתוכנה