
תוכנה 1 בשפת Java

שיעור מספר 3: "חתיכת טיפוס"

ליאור וולף

ליאור שפירא

בית הספר למדעי המחשב
אוניברסיטת תל אביב



על סדר היום

- מודל הזיכרון של Java – סיכום
- כלי עזר ב Java
- המחלקה כטיפוס נתונים
- שימוש בטיפוסי מחלקה קיימים
- הגדרת טיפוסים חדשים
- מודל הזיכרון של זימון שירותי מופע

גושי אתחול סטטיים

- ראינו כי אתחול המשתנה הסטטי התרחש מיד לאחר טעינת המחלקה לזיכרון, עוד לפני פונקציית ה `main`
- ניתן לבצע פעולות נוספות (בדרך כלל אתחולים למניהם) מיד לאחר טעינת המחלקה לזיכרון, פעולות אלו יש לציין בתוך בלוק `static`
- פרטים – באתר הקורס

תמונת הזיכרון האמיתית

- מודל הזיכרון שתואר כאן הוא פשטני – פרטים רבים נוספים נשמרים על המחסנית וב-Heap
- תמונת הזיכרון האמיתית והמדויקת היא תלוית סביבה ועשויה להשתנות בריצות בסביבות השונות
- נושא זה נידון בהרחבה בקורס "קומפילציה"



המחלקה כספריה של שרותים

- ניתן לראות במחלקה ספריה של שרותים, מודול: 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 השיעור
- ואולם קיימות ב- Java גם כמה מחלקות המשמשות כספריות בלבד. בין השימושיות שבהן:
 - `java.lang.Math`
 - `java.util.Arrays`
 - `java.lang.System`

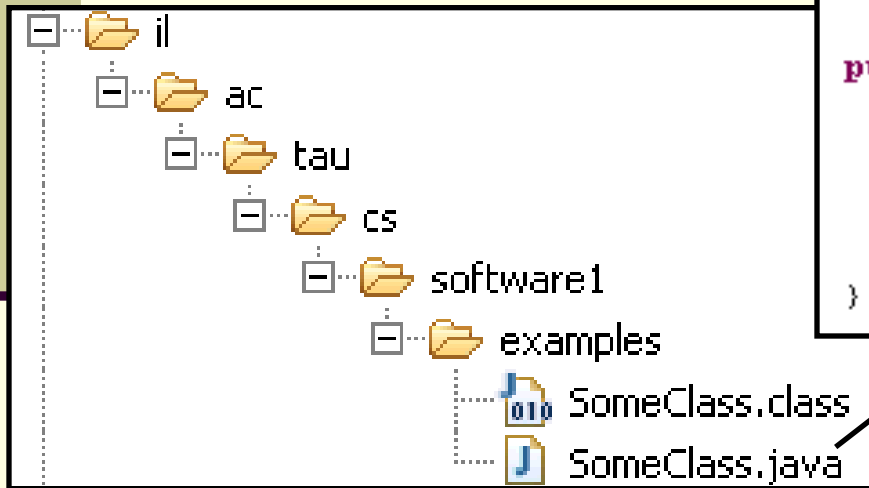


חבילות ומרחב השמות

- 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;
```

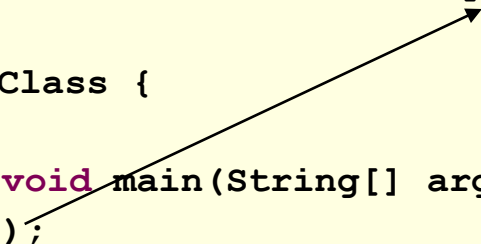
משפט static import

- 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 הקומפילר וגורר טעות קומפילציה ("התנגשות שמות")
- סביבות הפיתוח המודרניות יודעות לארגן בצורה אוטומטית את משפטי ה- `import` כדי להימנע מייבוא גורף מדי ("name pollution")



CLASSPATH

- איפה נמצאות המחלקות?
- איך יודעים הקומפיילר וה-JVM היכן לחפש את המחלקות המופיעות בקוד המקור 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 איתם?
- בעולם התוכנה מקובל לספק ביחד עם הספריות גם מסמך תיעוד, המפרט את שמות וחתימות 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 (**A**pplication **P**rogramming **I**nterface)
- תוכנת ה-`javadoc` היא חלק מה-`JDK` וניתן להשתמש בה משורת הפקודה או מתוך סביבת הפיתוח

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

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

```
public class SomeClass {
```

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

```
/** Documentaion 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

מחלקות כטיפוסי נתונים

מחלקות כטיפוסי נתונים

- ביסודה של גישת התכנות מונחה העצמים היא ההנחה שניתן לייצג ישויות מעולם הבעיה ע"י ישויות בשפת התכנות

- בכתיבת מערכת תוכנה בתחום מסוים (domain), נרצה לתאר את המרכיבים השונים באותו תחום כטיפוסיים ומשתנים בתוכנית המחשב

- התחומים שבהם נכתבות מערכות תוכנה מגוונים:

- בנקאות, ספורט, תרופות, מוצרי צריכה, משחקים ומולטימדיה, פיסיקה ומדע, מנהלה, מסחר ושרותים...

- יש צורך בהגדרת **טיפוסי נתונים** שישקפו את התחום, כדי שנוכל לעלות ברמת ההפשטה שבה אנו כותבים תוכניות

מחלקות כטיפוסי נתונים

- מחלקות מגדירות טיפוסים שהם הרכבה של טיפוסים אחרים (יסודיים או מחלקות בעצמם)
- מופע (instance) של מחלקה נקרא עצם (object)
- בשפת Java כל המופעים של מחלקות הם עצמים חסרי שם (אנונימיים) והגישה אליהם היא דרך הפניות בלבד
- כל מופע עשוי להכיל:
 - נתונים (data members, instance fields)
 - שרותים (instance methods)
 - פונקציות אתחול (בנאים, constructors)

מחלקות ועצמים

- כבר ראינו בקורס שימוש בטיפוסים שאינם פרימיטיביים: מחרוזת ומערך
 - גם ראינו שעקב שכיחות השימוש בהם יש להם הקלות תחביריות מסוימות (פטור מ- `new` והעמסת אופרטור)
- ראינו כי עבודה עם טיפוסים אלה מערבת שתי ישויות נפרדות:
 - **העצם**: המכיל את המידע
 - **ההפנייה**: משתנה שדרכו ניתן לגשת לעצם
- זאת בשונה ממשתנים יסודיים (טיפוסים פרימיטיביים)
- דוגמא: בהגדרה: `int i=5 , j=7;`
i ו- j הם מופעים של `int` כשם ש `"hello"` ו- `"world"` הם מופעים של `String`

שרותי מופע

■ למחלקות יש שרותי מופע – פונקציות אשר מופעלות על מופע מסוים של המחלקה

■ תחביר של הפעלת שרות מופע הוא:

```
objRef.methodName(arguments)
```

לדוגמא:

```
String str = "SupercaliFrajalistic";  
int len = str.length();
```

■ זאת בשונה מזימון שרות מחלקה (static):

```
className.methodName(arguments)
```

לדוגמא:

```
String.valueOf(15); // returns the string "15"
```

■ שימו של כי האופרטור נקודה (.) משמש בשני המקרים בתפקידים שונים לגמרי!

שימוש במחלקות קיימות

- לטיפוס מחלקה תכונות בסיסיות, אשר סיפק כותב המחלקה, ואולם ניתן לבצע עם העצמים פעולות מורכבות יותר ע"י שימוש באותן תכונות
- את התכונות הבסיסיות יכול הספק לציין למשל בקובץ תיעוד
- תיעוד נכון יתאר מה השרותים הללו עושים ולא איך הם ממומשים
- התיעוד יפרט את חתימת השרותים ואת החוזה שלהם
- נתבונן במחלקה Turtle המייצגת צב לוגו המתקדם על משטח ציור
 - כאשר זנבו למטה הוא מצייר קו במסלול ההתקדמות
 - כאשר זנבו למעלה הוא מתקדם ללא ציור
- כותב המחלקה לא סיפק את הקוד שלה אלא רק עמוד תיעוד המתאר את הצב (המחלקה ארוזה ב JAR של קובצי class)

Turtle API

Class Turtle

java.lang.Object
|
+--**Turtle**

public class **Turtle**
extends java.lang.Object

A Turtle is a logo turtle that is used to draw. a turtle has a pen attached to a tail. If the tail is down the turtle draws as it moves on the plane.

Constructor Summary

Turtle ()	constructs a new turtle
------------------	-------------------------

Method Summary

double	getAngle ()	returns the direction which the turtle is facing
static int	getDelay ()	return the delay the turtle
double	getX ()	returns the x coordinate of the turtle's location
double	getY ()	returns the y coordinate of the turtle's location
void	hide ()	hides this turtle
void	home ()	moves the turtle to it's initial location and orientation
boolean	isTailDown ()	
boolean	isVisible ()	
void	jumpTo (int newX, int newY)	moves the turtle to the given x,y location without drawing a line from the current location
static void	main (java.lang.String[] args)	

בנאי – פונקצית אתחול -
ניתן לייצר מופעים חדשים של
המחלקה ע"י קריאה לבנאי עם
האופרטור new

שרותים – נפריד בין 2 סוגים
שונים:

1. **שרותי מחלקה** – אינם
מתייחסים לעצם מסוים,
מסומנים static

2. **שרותי מופע** – שרותים אשר
מתייחסים לעצם מסוים.
יפנו לעצם מסוים ע"י שימוש
באופרטור הנקודה

Turtle API

Method Summary

double	<code>getAngle()</code> returns the direction which the turtle is facing
static int	<code>getDelay()</code> return the delay the turtle
double	<code>getX()</code> returns the x coordinate of the turtle's location
double	<code>getY()</code> returns the y coordinate of the turtle's location
void	<code>hide()</code> hides this turtle
void	<code>home()</code> moves the turtle to it's initial location and orientation
boolean	<code>isTailDown()</code>
boolean	<code>isVisible()</code>
void	<code>jumpTo(int newX, int newY)</code> moves the turtle to the given x,y location without drawing a line from the current location
static void	<code>main(java.lang.String[] args)</code>
void	<code>moveBackward(double units)</code> moves the turtle backwards by the given units.
void	<code>moveForward(double units)</code> moves the turtle forward by the given units.
void	<code>setAngle(double angle)</code> sets the angle of which the turtle is facing to the given angle
static void	<code>setDelay(int _delay)</code> sets the delay of the turtle motion in milliseconds - default delay is 0
void	<code>setVisible(boolean visible)</code> sets the visibility of the turtle
void	<code>show()</code> shows this turtle
void	<code>tailDown()</code> sets the turtle tail down
void	<code>tailUp()</code> sets the turtle tail up
void	<code>turnLeft(int degrees)</code> turns the turtle left by the given degrees
void	<code>turnRight(int degrees)</code> turns the turtle right by the given degrees

- סוגים של שרותי מופע:
- שאליות (queries) –
 - שרותים שיש להם ערך מוחזר
 - בדרך כלל לא משנים את מצב העצם
 - בשיעור הבא נדון בסוגים שונים של שאליות

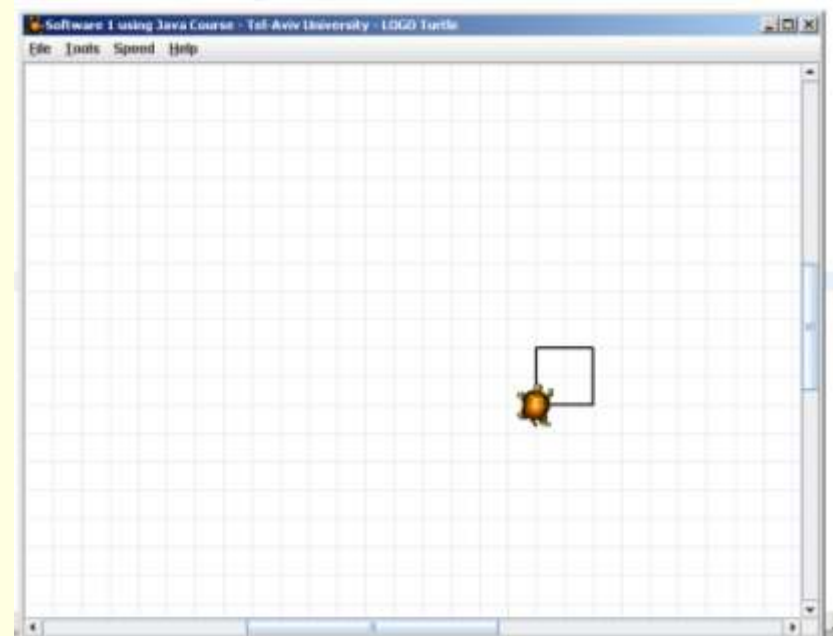
2. פקודות (commands) –

- שרותים ללא ערך מוחזר
- בדרך כלל משנים את מצב העצם שעליו הם פועלים

דוגמת שימוש

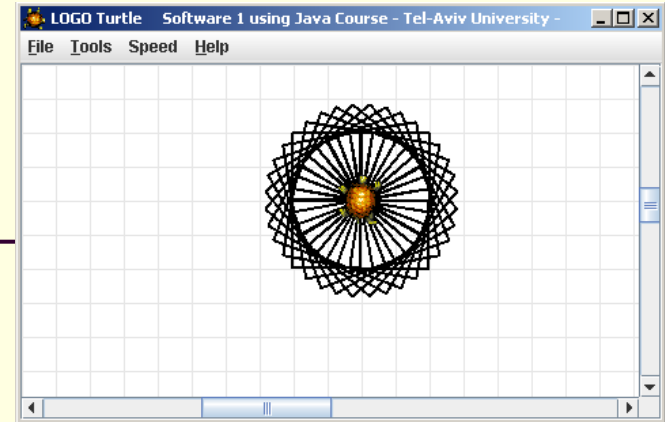


```
public class TurtleClient {  
  
    public static void main(String[] args) {  
        Turtle leonardo = new Turtle();  
  
        if(!leonardo.isTailDown())  
            leonardo.tailDown();  
  
        leonardo.moveForward(50);  
        leonardo.turnRight(90);  
  
        leonardo.moveForward(50);  
        leonardo.turnRight(90);  
  
        leonardo.moveForward(50);  
        leonardo.turnRight(90);  
  
        leonardo.moveForward(50);  
        leonardo.turnRight(90);  
  
    }  
}
```



עוד דוגמת שימוש

```
public class TurleClient {  
  
    public static void main(String[] args) {  
        Turtle leonardo = new Turtle();  
        leonardo.tailDown();  
        drawSquarePattern(leonardo, 50, 10);  
    }  
  
    public static void drawSquare(Turtle t, int size) {  
        for (int i = 0; i < 4; i++) {  
            t.moveForward(size);  
            t.turnRight(90);  
        }  
    }  
  
    public static void drawSquarePattern(Turtle t, int size, int angle) {  
        for (int i = 0; i < 360/angle; i++) {  
            drawSquare(t, size);  
            t.turnRight(angle);  
        }  
    }  
}
```

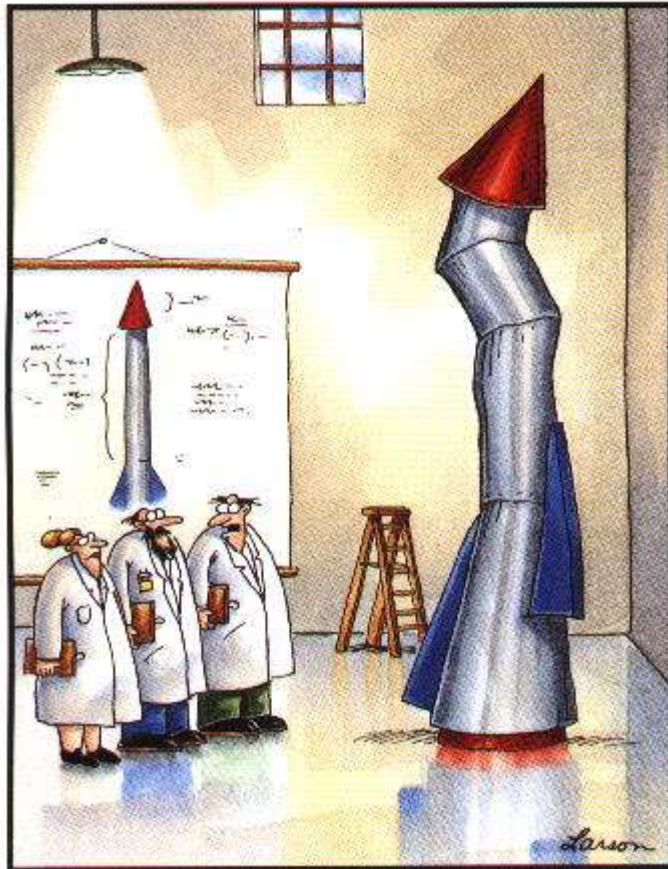




"לאונרדו יודע..."

- מה לאונרדו יודע לעשות ומה אנו צריכים ללמד אותו?
- מדוע המחלקה `Turtle` לא הכילה מלכתחילה את השרותים `drawSquare` ו-`drawSquarePattern` ?
 - יש לכך יתרונות וחסרונות
- איך לימדנו את הצב את התעלולים החדשים?
- נשים לב להבדל בין השרותים הסטטיים שמקבלים **עצם כארגומנט ומבצעים עליו פעולות** ובין שרותי המופע אשר אינם מקבלים את העצם **כארגומנט מפורש** (העצם מועבר מאחורי הקלעים)

הגדרת טיפוסים חדשים



"It's time we face reality, my friends. ...
We're not exactly rocket scientists."



The cookie cutter

- כאשר מכינים עוגיות מקובל להשתמש בתבנית ברזל או פלסטיק כדי ליצור עוגיות בצורות מעניינות (כוכבים)
- תבנית העוגיות (cookie cutter) היא מעין **מחלקה** ליצירת עוגיות
- העוגיות עצמן הן **מופעים** (עצמים) שנוצקו מאותה תבנית
- כאשר ה JVM טוען לזכרון את קוד המחלקה עוד לא נוצר אף **מופע** של אותה המחלקה. המופעים יוצרו בזמן מאוחר יותר – כאשר הלקוח של המחלקה יקרא מפורשות לאופרטור **new**
- ממש כשם שכאשר רכשת תבנית עוגיות עוד אין לך אף עוגייה
- לא ניתן לאכול את התבנית – רק עוגיות שנייצר בעזרתה!
- אנו אמנם יודעים מה תהיה **צורתן** של העוגיות העתידיות שיווצרו בעזרת התבנית אבל לא מה יהיה **טעמן** (שוקולד? וניל?)

דוגמא

■ נתבונן במחלקה MyDate לייצוג תאריכים:

```
public class MyDate {  
    int day;  
    int month;  
    int year;  
}
```

■ שימו לב! המשתנים `year`, `month` ו-`day` הוגדרו ללא המציין `static` ולכן בכל מופע עתידי של עצם מהמחלקה `MyDate` יופיעו 3 השדות האלה

■ שאלה: כאשר ה `JVM` טוען לזיכרון את המחלקה איפה בזיכרון נמצאים השדות `year`, `month` ו-`day`?

■ תשובה: הם עוד לא נמצאים! הם ייווצרו רק כאשר לקוח ייצר מופע (עצם) מהמחלקה

לקוח של המחלקה MyDate

- לקוח של המחלקה הוא קטע קוד המשתמש ב- MyDate
- למשל: כנראה שמי שכותב יישום של יומן פגישות צריך להשתמש במחלקה
- דוגמא:

```
public class MyDateClient {  
  
    public static void main(String[] args) {  
        MyDate d1 = new MyDate();  
  
        d1.day = 29;  
        d1.month = 2;  
        d1.year = 1984;  
  
        System.out.println(d1.day + "/" + d1.month + "/" + d1.year);  
    }  
}
```

- בדוגמא אנו רואים:
 - שימוש באופרטור ה- new ליצירת מופע חדש מטיפוס MyDate
 - שימוש באופרטור הנקודה לגישה לשדה של המופע המוצבע ע"י d1

אם שרות, אז עד הסוף

- האם התאריך `d1` מייצג תאריך תקין?
- מה יעשה כותב היומן כאשר יצטרך להזיז את הפגישה בשבוע?
- האם `d1.day += 7` ?
- כמו כן, אם למחלקה כמה לקוחות שונים – אזי הלוגיקה הזו תהיה משוכפלת אצל כל אחד מהלקוחות
- אחריותו של מי לוודא את תקינות התאריכים ולממש את הלוגיקה הנלווית?
- המחלקה היא גם מודול. אחריותו של הספק – כותב המחלקה – לממש את כל הלוגיקה הנלווית לייצוג תאריכים
- כדי לאכוף את עקביות המימוש (משתמר המחלקה) על משתני המופע להיות פרטיים

```

public class MyDate {

    private int day;
    private int month;
    private int year;

    public static void incrementDate(MyDate d) {
        // changes d to be the consequent day
    }

    public static String toString(MyDate d) {
        return d.day + "/" + d.month + "/" + d.year;
    }

    public static void setDay(MyDate d, int day) {
        /* changes the day part of d to be day if
        * the resulting date is legal */
    }

    public static int getDay(MyDate d) {
        return d.day;
    }

    private static boolean isLegal(MyDate d) {
        // returns if d represents a legal date
    }

    // more...
}

```

כדי להכליל את הדוגמא נחליף
 את שם המשתנה d שמסמן
 date ב- this שיסמל עצם
 מטיפוס כלשהו

```
public class MyDate {
```

```
private int day;  
private int month;  
private int year;
```

```
public static void incrementDate(MyDate this) {  
    // changes d to be the consequent day  
}
```

```
public static String toString(MyDate this) {  
    return this.day + "/" + this.month + "/" + this.year;  
}
```

```
public static void setDay(MyDate this, int day) {  
    /* changes the day part of d to be day if  
     * the resulting date is legal */  
}
```

```
public static int getDay(MyDate this) {  
    return this.day;  
}
```

```
private static boolean isLegal(MyDate this) {  
    // returns if d represents a legal date  
}
```

```
// more...
```

```
}
```

שימו לב!
השימוש במילה this כמציין שם של משתנה הוא אסור!
המילה this היא מילה שמורה בשפת java ואסורה לשימוש עבור משתני משתמש.

בהמשך הדוגמא יובהר
מדוע בחרנו דווקא
להשתמש בשם זה

נראות פרטית

מכיוון שהשדות `year`, `month` ו-`day` הוגדרו בנראות פרטית (private) לא ניתן להשתמש בהם מחוץ למחלקה (שגיאת קומפילציה)

```
public class MyDateClient {  
  
    public static void main(String[] args) {  
        MyDate d1 = new MyDate();  
  
        ❌ d1.day = 29;  
        ❌ d1.month = 2;  
        ❌ d1.year = 1984;  
    }  
}
```

כדי לשנות את ערכם יש להשתמש בשרותים הציבוריים שהוגדרו לשם כך

לקוח של המחלקה MyDate

```
public class MyDateClient {  
  
    public static void main(String[] args) {  
        MyDate d1 = new MyDate();  
  
        MyDate.setDay(d1, 29);  
        MyDate.setMonth(d1, 2);  
        MyDate.setYear(d1, 1984);  
  
        System.out.println(MyDate.toString(d1));  
    }  
}
```

כעת הדוגמא מתקמפלת אך עדיין נותרו בה שתי בעיות:

- השימוש בפונקציות גלובליות (סטטיות) מסורבל
- עבור כל פונקציה אנו צריכים להעביר את `d1` כארגומנט
- מיד לאחר השימוש באופרטור ה `new` קיבלנו עצם במצב לא עיקבי
- עד לביצוע השמת התאריכים הוא מייצג את התאריך הלא חוקי `0/0/00`

שרותי מופע

- כדי לפתור את הבעיה הראשונה, נשתמש בסוג שני של שרותים הקיים ב Java – שרותי מופע
- אלו הם שרותים המשויכים למופע מסוים – הפעלה שלהם נחשבת כבקשה או שאלה מעצם מסוים – והיא מתבצעת בעזרת אופרטור הנקודה
- בגלל שהבקשה היא מעצם מסוים, אין צורך להעביר אותו כארגומנט לפונקציה
- מאחורי הקלעים הקומפיילר מייצר משתנה בשם `this` ומעביר אותו לפונקציה, ממש כאילו העביר אותו המשתמש בעצמו

ממתקים להמונים

■ ניתן לראות בשרותי מופע סוכר תחבירי לשרותי מחלקה

■ ניתן לדמיין את שרות המופע `m()` של מחלקה `C` כאילו היה שרות מחלקה (סטטי) המקבל עצם מהטיפוס `C` כארגומנט:

```
public class C {
```

```
    public void m(args) {
```

```
        ...
```

```
    }
```

```
}
```

```
public static void m(C this, args) {
```

```
    ...
```

```
}
```



ממתקים להמונים

■ בראייה זו, הקריאות למתודה `m()` של לקוחות המחלקה `C` יתורגמו ע"י העברת ההפניה שעליה בוצעה הקריאה כארגומנט לשרות הסטטי:

```
public class SomeClient {  
  
    public static void main(String[] args) {  
        C obj = new C();  
        obj.m(args);  
    }  
}
```

`C.m(obj, args)`

"לא מה שחשבת"

- שרותי מופע מספקים תכונה נוספת ל Java פרט לסוכר התחבירי
- בהמשך הקורס נראה כי לשרותי המופע ב Java תפקיד מרכזי בשיגור שרותים דינאמי (dynamic dispatch), תכונה בשפה המאפשרת החלפת המימוש בזמן ריצה ופולימורפיזם
- תאור שרותי מופע כסוכר תחבירי הוא פשטני (ושגוי!) אך נותן אינטואיציה טובה לגבי פעולת השרות בשלב זה של הקורס

```

public class MyDate {

    private int day;
    private int month;
    private int year;

    public void incrementDate( ) {
        // changes itself to be the consequent day
    }

    public String toString( ) {
        return day + "/" + month + "/" + year;
    }

    public void setDay( int day) {
        /* changes the day part of itself to be day if
        * the resulting date is legal */
    }

    public int getDay( ) {
        return day;
    }

    private boolean isLegal( ) {
        // returns if the argument represents a legal date
    }

    // more...
}

```

```

public class MyDate {

    private int day;
    private int month;
    private int year;

    public void incrementDate(){
        // changes current object to be the consequent day
    }

    public String toString(){
        return day + "/" + month + "/" + year;
    }

    public void setDay(int day){
        /* changes the day part of the current object to be day if
        * the resulting date is legal */
    }

    public int getDay(){
        return day;
    }

    private boolean isLegal(){
        // returns if the current object represents a legal date
    }

    // more...
}

```



בנאים

- כדי לפתור את הבעיה שהעצם אינו מכיל ערך תקין מיד עם יצירתו נגדיר עבור המחלקה **בנאי**
- **בנאי** הוא **פונקציה אתחול** הנקראת ע"י אופרטור ה **new** מיד אחרי שהוקצה מקום לעצם החדש. שמה כשם המחלקה שהיא מאתחלת וחתימתה אינה כוללת ערך מוחזר
- זיכרון המוקצה על ה-Heap (למשל ע"י **new**) מאותחל אוטומטית לפי הטיפוס שהוא מאכסן (0, null, false), כך שאין צורך לציין בבנאי אתחול שדות לערכים אלה
- המוטיבציה המרכזית להגדרת בנאים היא הבאת העצם הנוצר למצב שבו הוא מקיים את משתמר המחלקה וממופה למצב מופשט בעל משמעות (יוסבר בהמשך)

```

public class MyDate {

    public MyDate(int day, int month, int year) {
        this.day = day;
        this.month = month;
        this.year = year;
    }

    // ...
}

```

הגדרת בנאי ל MyDate

```

public class MyDateClient {

    public static void main(String[] args) {
        MyDate d1 = new MyDate(29,2,1984);
        d1.incrementDate();

        System.out.println(d1.toString());
    }
}

```

קוד לקוח המשתמש ב- MyDate

מודל הזיכרון של זימון שרותי מופע

מודל הזיכרון של זימון שרותי מופע

■ בדוגמא הבאה נראה כיצד מייצר הקומפיילר עבורנו את ההפניה `this` עבור כל בנאי וכל שרות מופע

■ נתבונן במחלקה `Point` המייצגת נקודה במישור הדו מימדי. כמו כן המחלקה מנהלת מעקב בעזרת משתנה גלובלי (סטטי) אחר מספר העצמים שנוצרו מהמחלקה

■ בהמשך הקורס נציג מימוש מלא ומעניין יותר של המחלקה, אולם כעת לצורך פשטות הדוגמא נסתפק בבנאי, שדה מחלקה, 2 שדות מופע ו-3 שרותי מופע

```

public class Point {

    private static double numOfPoints;

    private double x;
    private double y;

    public Point(double x, double y){
        this.x = x;
        this.y = y;
        numOfPoints++;
    }

    public double getX() {
        return x;
    }

    /** tolerant method, no precondition - for nonresponsible clients
     * @post (newX > 0.0 && newX < 100.0) $implies getX() == newX
     * @post !(newX > 0.0 && newX < 100.0) $implies getX() == $prev(getX())
     */
    public void setX(double newX) {
        if(newX > 0.0 && newX < 100.0)
            doSetX(newX);
    }

    /** only business logic. Has a precondition - for responsible clients
     * @pre (newX > 0.0 && newX < 100.0)
     * @post getX() == newX
     */
    public void doSetX(double newX) {
        x = newX;
    }

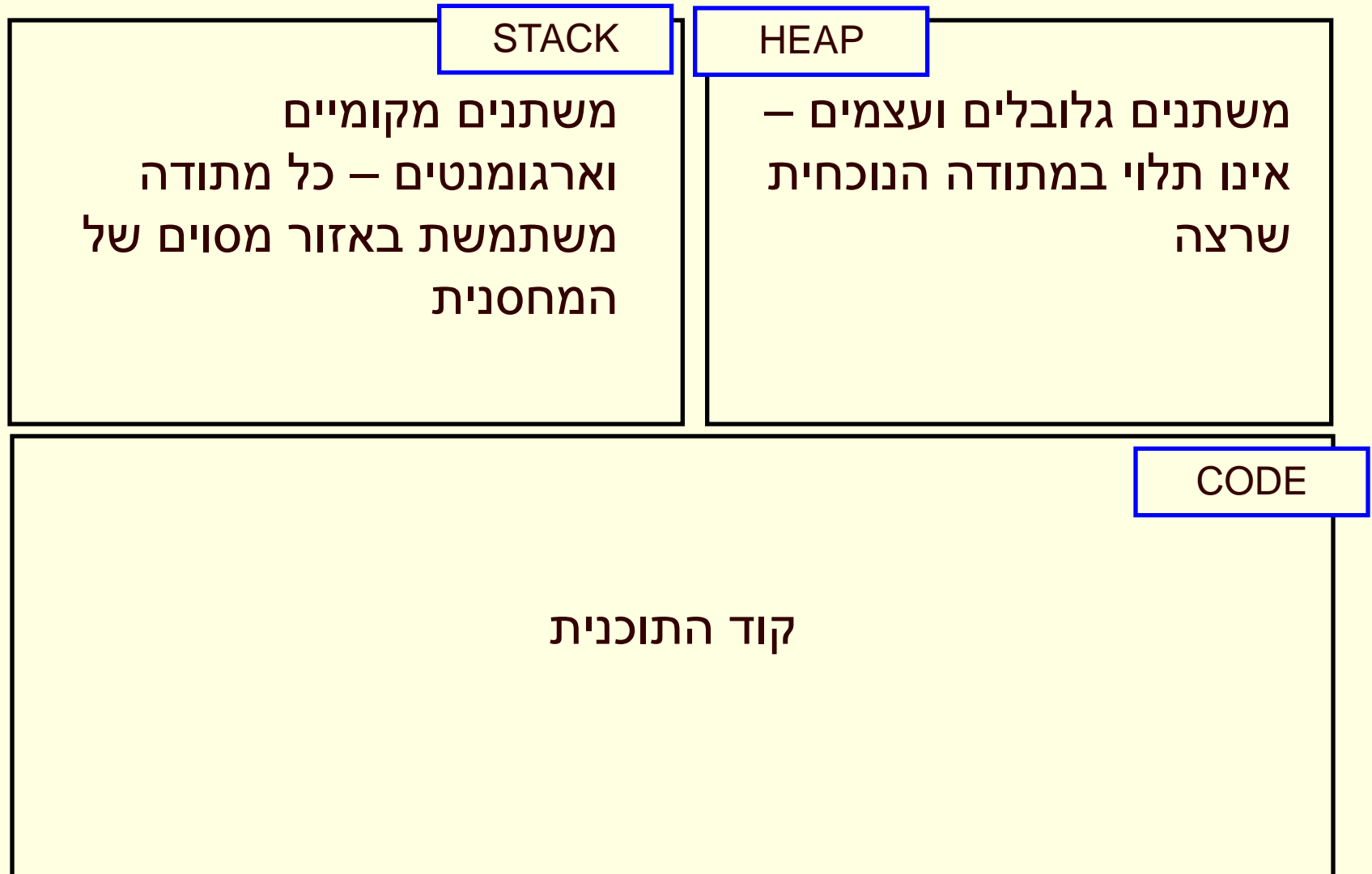
    // More methods...
}

```

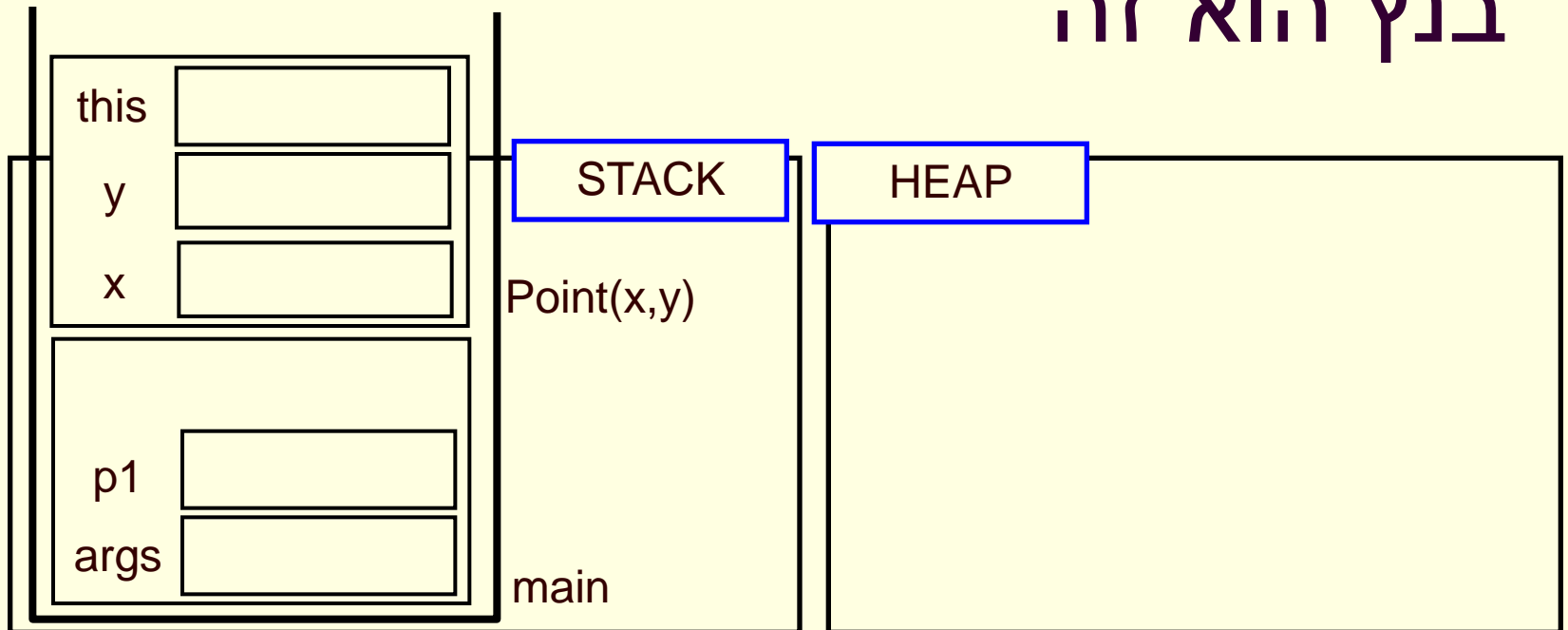

PointUser

```
public class PointUser {  
  
    public static void main(String[] args) {  
        Point p1 = new Point(1.0, 2.0);  
        Point p2 = new Point(10.0, 20.0);  
  
        p1.setX(11.0);  
        p2.setX(21.0);  
  
        System.out.println("p1.x == " + p1.getX());  
    }  
  
}
```

מודל הזיכרון של Java



"בנוץ הוא זה"



```
public class PointUser {  
  
    public static void main(String[] args) {  
        Point p1 = new Point(1.0, 2.0);  
        Point p2 = new Point(10.0, 20.0);  
  
        p1.setX(11.0);  
        p2.setX(21.0);  
  
        System.out.println("p1.x == "  
                            + p1.getX());  
    }  
}
```

```
public class Point {  
  
    public Point(double x, double y){  
        this.x = x;  
        this.y = y;  
        numOfPoints++;  
    }  
  
    public double getX()  
    { return x; }  
  
    public void setX(double newX) {  
        if(newX > 0.0 && newX < 100.0)  
            doSetX(newX);  
    }  
  
    public void doSetX(double newX)  
    { x = newX; }  
}
```

CODE

סיכום ביניים

- **שרותי מופע** (instance methods) בשונה משרותי מחלקה (static method) פועלים על עצם מסוים (this) ■ בעוד ששרותי מחלקה פועלים בדרך כלל על הארגומנטים שלהם
- **משתני מופע** (instance fields) בשונה ממשתני מחלקה (static fields) הם **שדות בתוך עצמים**. הם נוצרים רק כאשר נוצר עצם חדש מהמחלקה (ע"י new)
- בעוד ששדות מחלקה הם משתנים גלובליים. קיים עותק אחד שלהם, שנוצר בעת טעינת קוד המחלקה לזכרון, ללא קשר ליצירת עצמים מאותה המחלקה

הסוף (כנראה)

עוד על עצמים ותמונת הזיכרון

```
public class BOOK1 {  
    private String title;  
    private int date;  
    private int page_count;  
}
```



<i>title</i>	"The Red and the Black"
<i>date</i>	1830
<i>page_count</i>	341

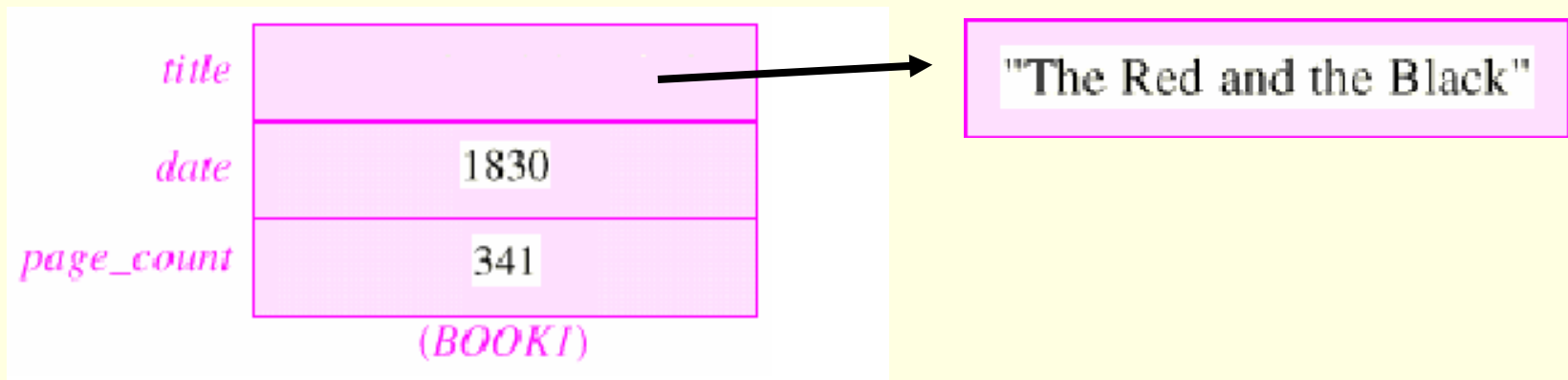
(BOOK1)

התרשים פשטני –
מחרוזת היא עצם ולכן
השדה title מכיל רק
הפנייה אליו

Simple Book



```
public class BOOK1 {  
    private String title;  
    private int date;  
    private int page_count;  
}
```



Writer Class

```
public class WRITER {  
    private String name;  
    private String real_name;  
    private int birth_year;  
    private int death_year;  
}
```



<i>name</i>	"Stendhal"
<i>real_name</i>	"Henri Beyle"
<i>birth_year</i>	1783
<i>death_year</i>	1842

(WRITER)

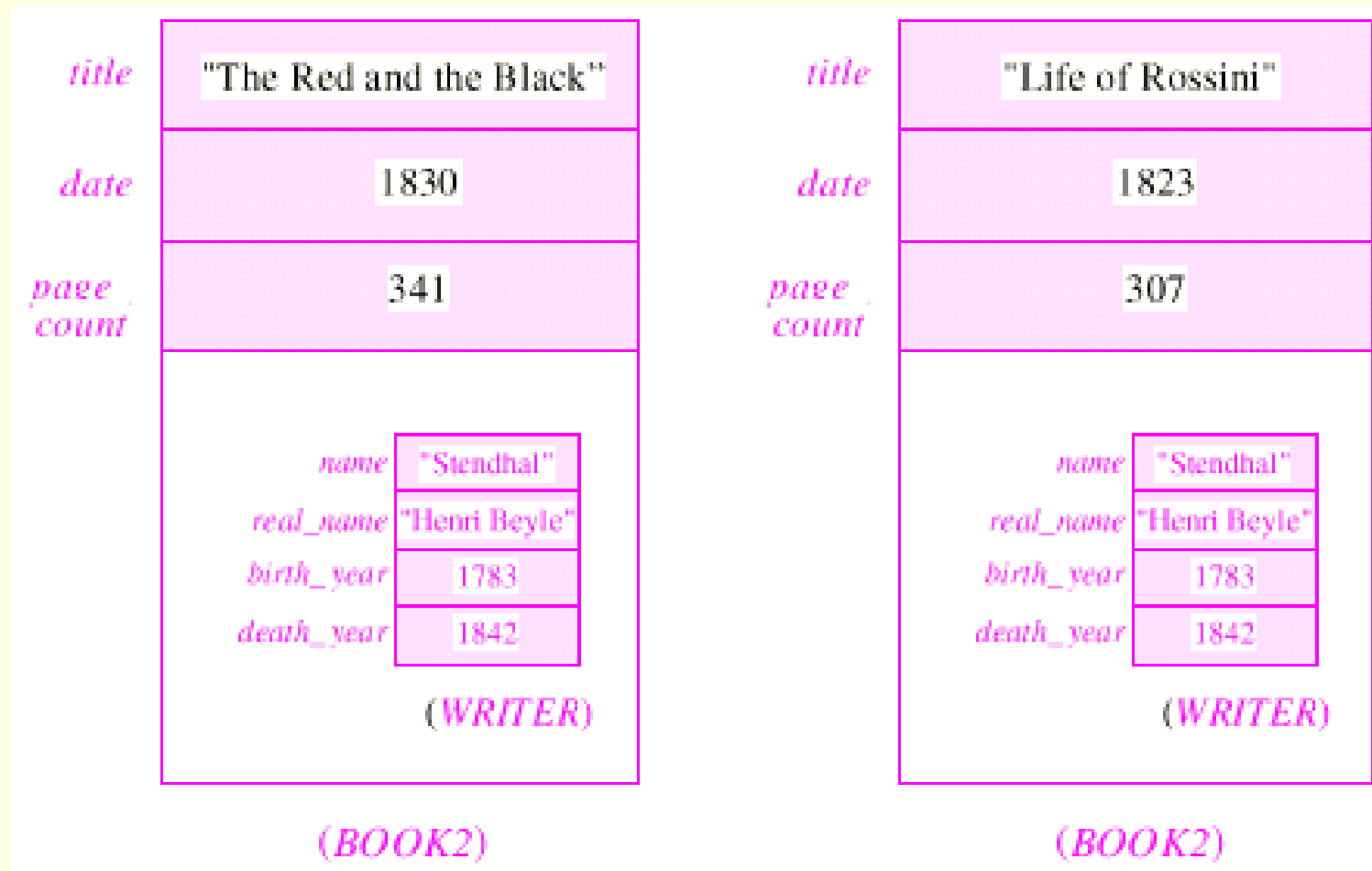
עצמים המתייחסים לעצמים

■ איך נבטא את הקשר שבין ספר ומחברו?

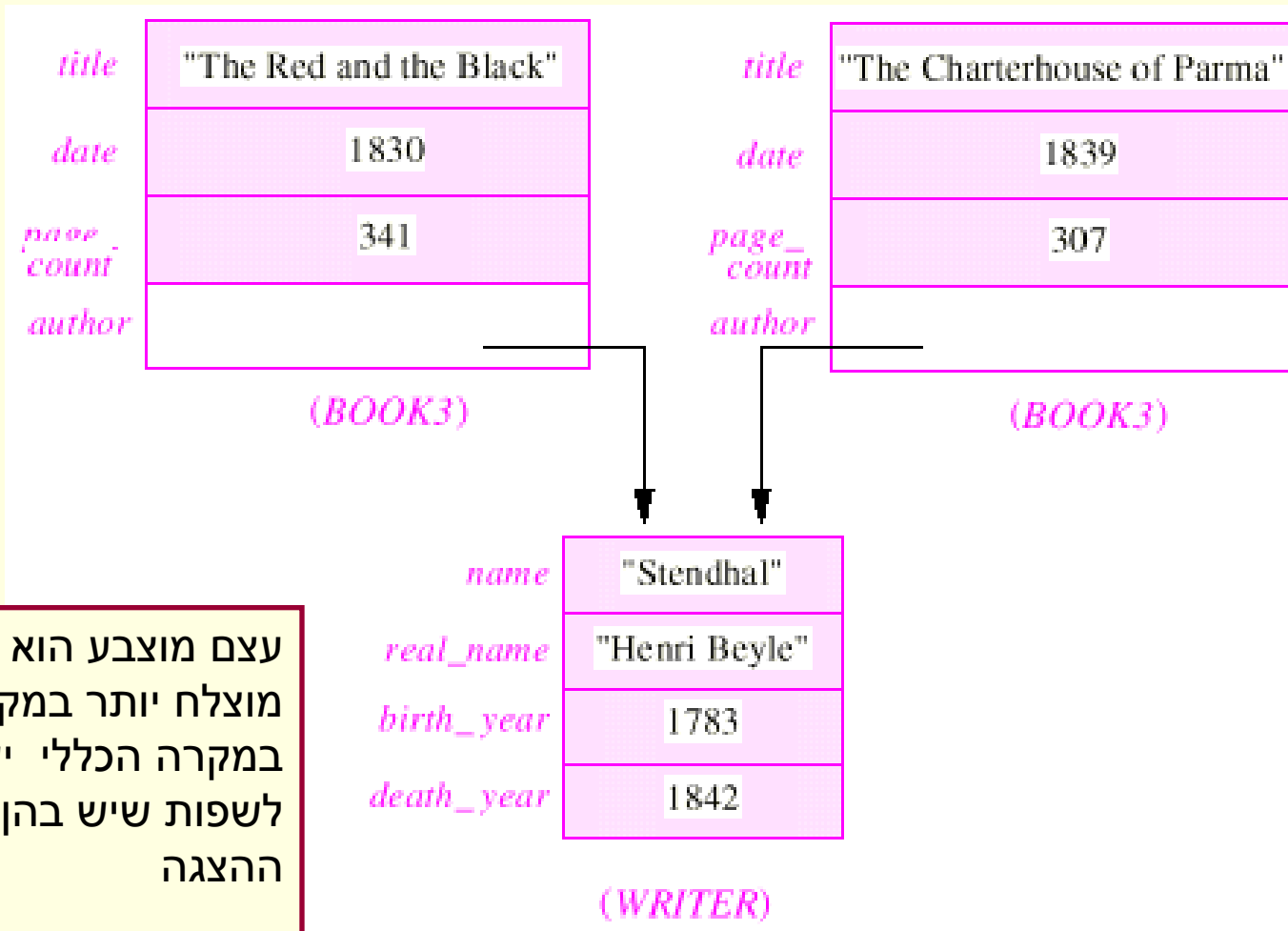
```
public class BOOK3 {  
    private String title;  
    private int date;  
    private int page_count;  
    private Writer author;  
}
```

■ בשפות תכנות אחרות (לא ב-Java) ניתן לבטא יחס זה בשתי דרכים שונות, שלכל אחת מהן השלכות על המודל

עצם מוכל (לא ב-Java)



עצם מוצבע



עצם מוצבע הוא כנראה רעיון מוצלח יותר במקרה זה, אולם במקרה הכללי יש יתרונות לשפות שיש בהן שתי צורות ההצגה

יתרונות העצם המוכל

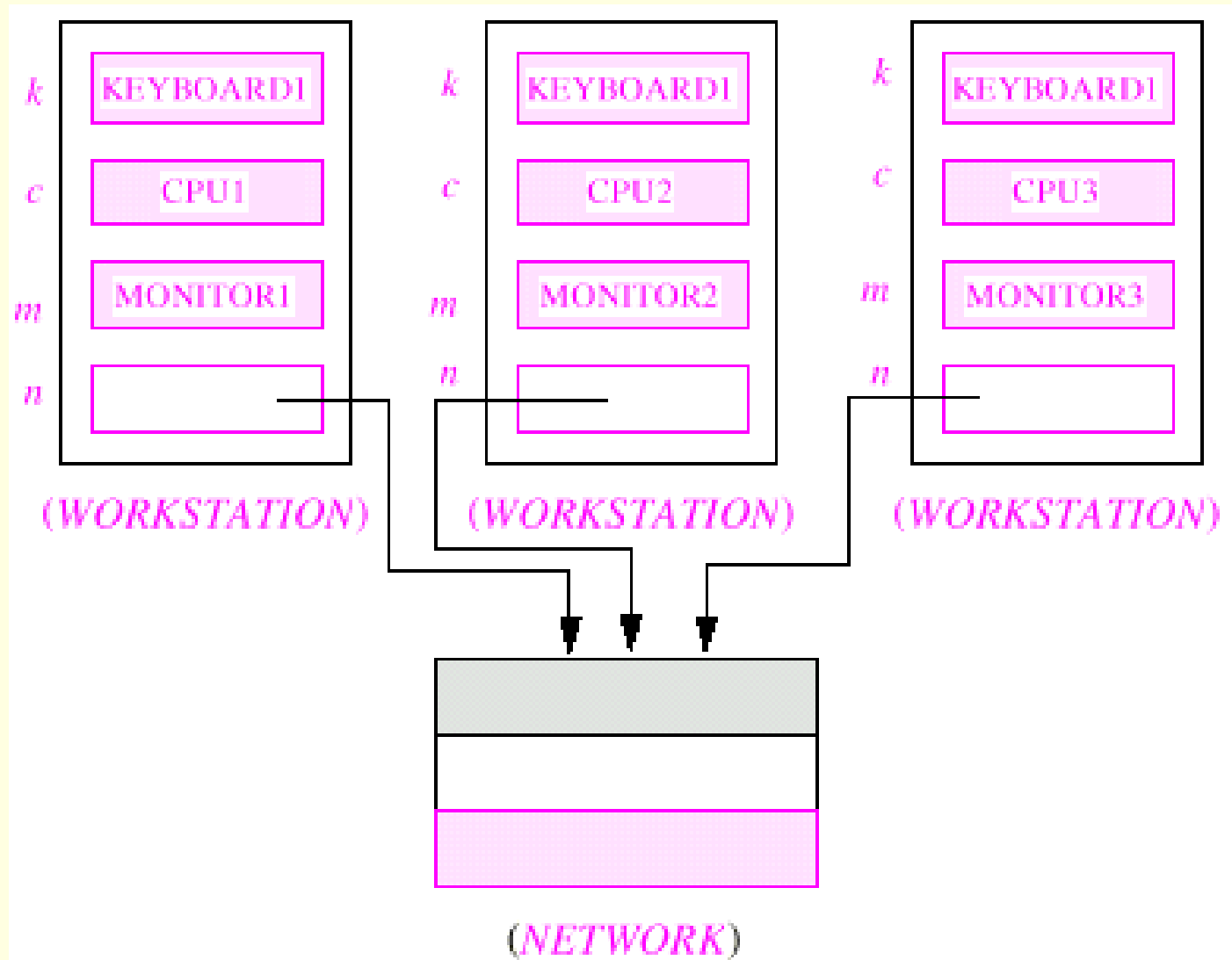
יעילות

- גישה לשדות מוכלים שלא דרך dereference של מצביע
- **מודל טוב יותר** – בהתאם למה שברצוננו לבטא
- מצביע למחלקה S פירושו שהלקוח "יודע על" S
 - עצם מוכל מעיד על כך שהלקוח מכיל S
 - בפרט, הכלה מרמזת על אי-שיתוף

תמיכה אחידה בטיפוסים פרימיטיביים

- עצמים מכילים את הטיפוסים היסודיים עצמם ולא מצביע אליהם

אובייקטים מורכבים – מודל טוב יותר



הכלה או הצבעה?

- בשפת Java הוחלט **שלא לאפשר** הכלת עצמים
- כל ההתייחסויות לעצמים בשפה הן הפניות
- הדבר מצריך משנה זהירות במקרים של **שיתוף** עצמים (sharing, aliasing)
- ניתן להתמודד עם קושי זה בעזרת אכיפה של **קיבעון** (immutability) כפי שנראה בשיעור הבא

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
- 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

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





חזרה אפשרי ל- divide

```
/**  
 * @pre denominator != 0 ,  
 *     "Can't divide by zero"  
 *  
 * @post Math.abs($ret * denominator) <= Math.abs( numerator ) ,  
 *     "always truncates the fraction"  
 *  
 * @post (($ret * denominator) + (numerator % denominator)) == numerator,  
 *     "regular divide"  
 */  
public static int divide(int numerator, int denominator)
```

■ התחביר מבוסס על כלי בשם Jose
■ לפעמים החזרה ארוך יותר מגוף הפונקציה



חזרה אפשרי אחר ל- divide

```
/**
 * @pre (denominator != 0) || (numerator != 0) ,
 *      "you can't divide zero by zero"
 *
 * @post (denominator == 0) && ((numerator > 0)) $implies
 *      $ret == Integer.MAX_VALUE
 *      "Dividing positive by zero yields infinity (MAX_INT)"
 *
 * @post (denominator == 0) && ((numerator < 0)) $implies
 *      $ret == Integer.MIN_VALUE
 *      "Dividing negative by zero yields minus infinity (MIN_INT)"
 *
 * @post Math.abs($ret * denominator) <= Math.abs(numerator) ,
 *      "always truncates the fraction"
 *
 * @post (denominator != 0) $implies
 *      (($ret * denominator)+(numerator % denominator)) == numerator,
 *      "regular divide"
 */
public static int divide(int numerator, int denominator)
```

תנאי קדם סובלניים מסבכים את מימוש הפונקציה - כפי שמתבטא בחלוקה



החזזה והמצב

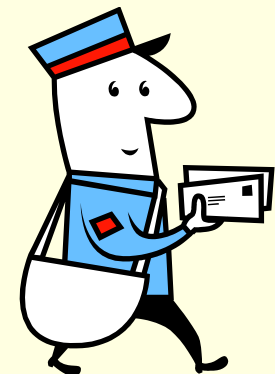
- חזזה של שרות אינו כולל רק את הארגומנטים שלו
- תנאי קדם של חזזה יכול להגדיר **מצב** (תמונת זיכרון, קשירת ערכי משתנים) שרק בו ניתן לקרוא לפונקציה
- לדוגמא: במחלקה מסוימת קיימים שרות **המאתחל** מבנה נתונים ושרות **הקורא** מאותו מבנה נתונים (שדה מחלקה)
- תנאי הקדם של שרות הקריאה יכול להיות שמבנה הנתונים כבר אותחל ושנתרו בו הודעות
- נשים לב שמימוש `getNextMessage` מתעלם לחלוטין מהמקרים שבהם תנאי הקדם אינו מתקיים
- המימוש לא בודק את תנאי הקדם בגוף המתודה

הדואר בא היום

```
public static String [] messages = new String[INBOX_CAPACITY];
public static int head = 0;
public static boolean isIntialized = false;

public static void init(String login, String password){
    // connect to mail server...
    // put new messages on the messages array...
    // update head
    isIntialized = true;
}

/**
 * @pre isIntialized , "you must be logged in first"
 * @pre head < messages.length , "more messages to read"
 * @post "returns the next unread message"
 */
public static String getNextMessage(){
    return messages[head++];
}
```



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...

דוגמא

```
/**
 * @param a An 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 searchSorted(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 `searchSorted` לא לבדוק את תנאי הקדם. אם לקוח יפר אותו היא עלולה להחזיר ערך שגוי או אפילו לא להסתיים אבל זו כבר לא אשמתה...



חיזוק תנאי האחר

■ אם תנאי הקדם לא מתקיים, לשירות מותר שלא לקיים את תנאי האחר כשהוא מסיים; קריאה לשירות כאשר תנאי הקדם שלו לא מתקיים מהווה תקלה שמעידה על פגם בתוכנית

■ אבל גם אם תנאי הקדם לא מתקיים, מותר לשירות לפעול ולקיים את תנאי האחר

■ לשירות מותר גם לייצר כאשר הוא מסיים מצב הרבה יותר ספציפי מזה המתואר בתנאי האחר; תנאי האחר לא חייב לתאר בדיוק את המצב שיווצר אלא מצב כללי יותר (תנאי חלש יותר)

■ למשל, שירות המתחייב לביצוע חישוב בדיוק של ϵ כלשהו יכול בפועל להחזיר חישוב בדיוק של $\epsilon / 2$

דע מה אתה מבקש

■ מי מונע מאיתנו לעשות שטויות?

■ אף אחד

■ קיימים כלי תוכנה אשר מחוללים קוד אוטומטי, שיכול
לאכוף את קיום החוזה בזמן ריצה ולדווח על כך

■ השימוש בהם עדיין לא נפוץ

■ אולם, לציון החוזה (אפילו כהערה!) חשיבות
מתודולוגית נכבדה בתהליך תכנון ופיתוח מערכות
תוכנה גדולות

החוזה והקומפיילר

- יש הבטים מסויימים ביחס שבין ספק ללקוח שהם באחריותו של הקומפיילר
 - למשל: הספק לא צריך לציין בחוזה שהוא מצפה ל-2 ארגומנטים מטיפוס `int`, מכיוון שחתימת המתודה והקומפיילר מבטיחים זאת
- ספק לא יודע באילו הקשרים (`context`) יקראו לו
 - מי יקרא לו, עם אילו ארגומנטים, מה יהיה ערכם של משתנים גלובלים מסויימים ברגע הקריאה
 - רבים מההקשרים יתבררו רק בזמן ריצה
- הקומפיילר יודע לחשב רק מאפיינים סטטיים (כגון התאמת טיפוסים)
- לכן תנאי הקדם של החוזה יתמקדו בהקשרי הקריאה לשרות
 - ערכי הארגומנטים
 - ערכי משתנים אחרים ("המצב של התוכנית")