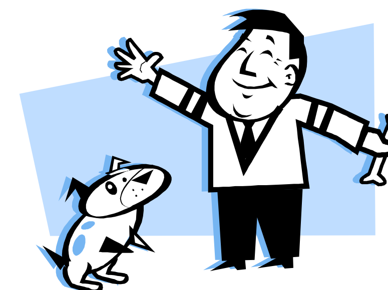
תוכנה 1 בשפת Java
שיעור מספר 3: חוזה

מועבר ע"י ליאור וולף 2008



בית הספר למדעי המחשב
אוניברסיטת תל אביב

על סדר היום

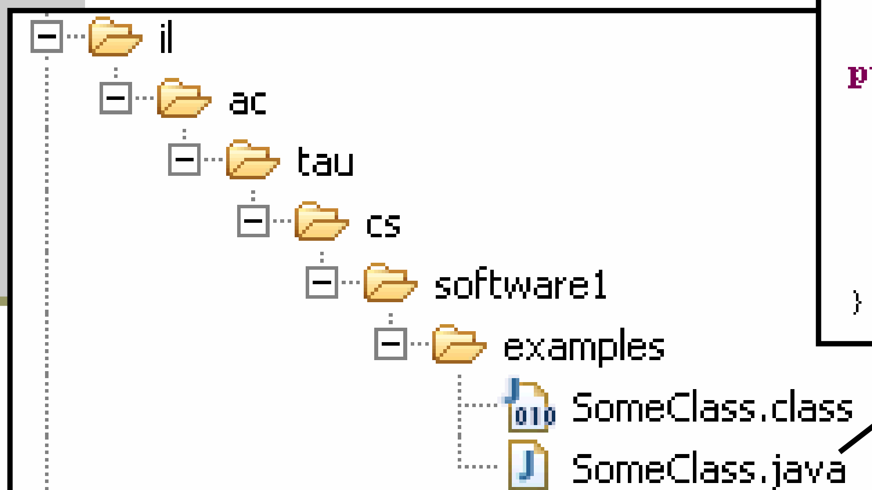
- חבילות, תיעוד
- חוזים, נכונות והסתרת מידע
- מחלקות כטיפוסי נתונים
- ייצוג עצמים בזיכרון התוכנית

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 משתמשת 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)




```java
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:

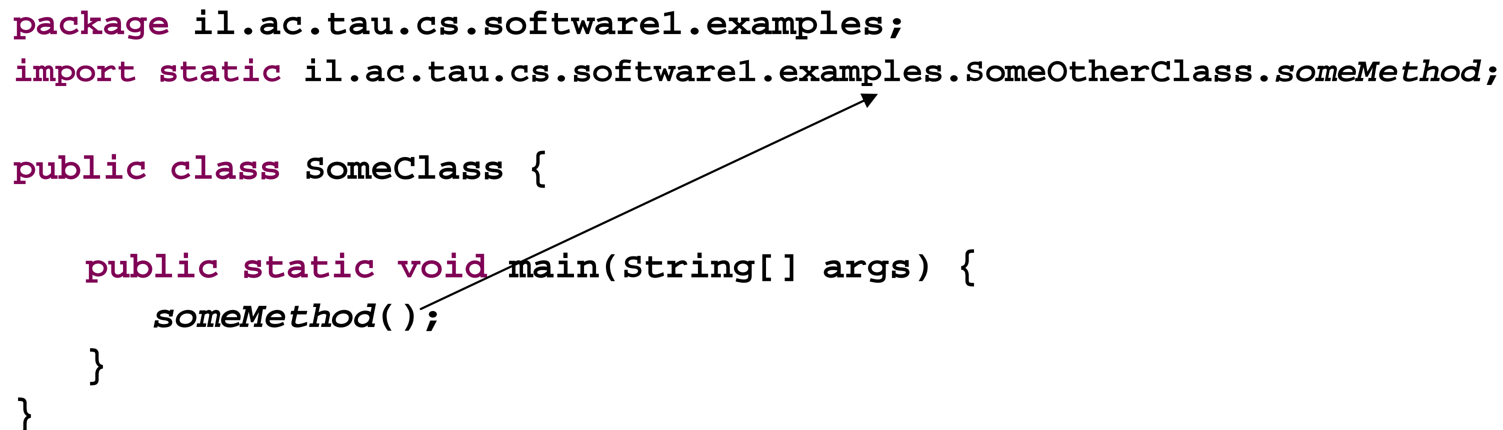
```
import java.util.*;
...
    System.out.println("Before: x=" + Arrays.toString(arr));
```

- 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:

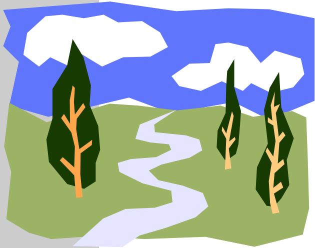


```
package il.ac.tau.cs.software1.examples;
import static il.ac.tau.cs.software1.examples.SomeOtherClass.someMethod;

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 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** (**J**ava **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 (**A**pplication **P**rogramming **I**nterface)
- תוכנת ה `javadoc` היא חלק מה- JDK וניתן להשתמש בה משורת הפקודה או מתוך סביבת הפיתוח

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

/** Documetntaion for the class
 * @author your name here
 */
public class SomeClass {

    /** Documetntaion for the field */
    public int someField;

    /** Documetntaion for the method
     * @param x documentation for parameter x
     * @param y documentation for parameter y
     * @return
     *     documentation for return value
     */
    public 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)
{...}
```

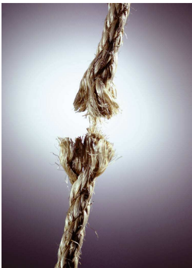
- לרוב הקוראים יש מושג כללי נכון לגבי הפונקציה ופעולתה
- למשל, די ברור מה תחזיר הפונקציה אם נקרא לה עם הארגומנטים 6 ו- 2

"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

חוזה אפשרי ל- divide

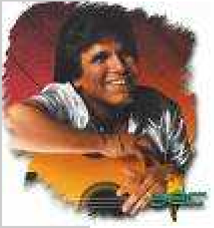
```
/**
 * @pre denominator != 0 ,
 *       "Can't divide by zero"
 *
 * @post Math.abs($ret * denominator) <= Math.abs(numerator) ,
 *        "always truncates the fraction"
 *
 *  @post (($ret * denominator) + (numerator % denominator)) == numerator,
 *        "regular divide"
 */
public static int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

- התחביר מבוסס על כלי בשם Jose (ודומה לכלים אחרים)
- לפעמים החוזה ארוך יותר מגוף הפונקציה

חוזה 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 שמשתקף בחוזה

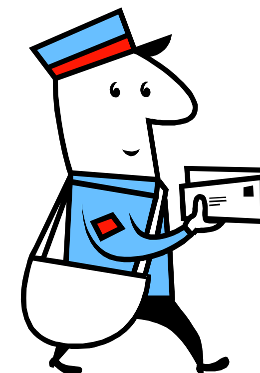


החוזה והמצב

- חוזה של שרות אינו כולל רק את הארגומנטים שלו
- תנאי קדם של חוזה 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 ושנותרו בו הודעות
- נשים לב שמימוש **getNextMessage** מתעלם לחלוטין מהמקרים שבהם תנאי הקדם אינו מתקיים
- המימוש לא בודק את תנאי הקדם בגוף המתודה
- באופן דומה לגבי שרותי מופע שתלויים במצב העצם – ערכי שדות המופע

הדואר בא היום

```java
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

```java
/**
 * @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
 *         it x does not occur in a
 *
 * @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 שייווצר אלא מצב כללי יותר (תנאי חלש יותר)
    - למשל, שירות המתחייב לביצוע חישוב בדיוק של ε כלשהו יכול בפועל להחזיר חישוב בדיוק של $\varepsilon/2$

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 מסוימים 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 מסוימים 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")

טענות על המצב

- 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 **הדו משמעיות** של השפה הטבעית וכן היא לא מניחה דבר על **אופן השימוש** בתוכנה

זהו אינו "דיון אקדמי"

- להוכחת נכונות של תוכנה חשיבות גדולה במגוון רחב של יישומים

- לדוגמא:
  - בתוכנית אשר שולטת על בקרת הכור הגרעיני נרצה שיתקיים בכל רגע נתון:

```
plutoniumLevel < CRITICAL_MASS_THRESHOLD
```

  - בתוכנית אשר שולטת על בקרת הטיסה של מטוס נוסעים נרצה שיתקיים בכל רגע נתון:

```
(cabinAirPressure < 1)
        $implies airMaskState == DOWN
```

- נרצה להשתכנע כי בכל רגע נתון בתוכנית לא יתכן כי המשתמר אינו **true**

הוכחת נכונות של טענה

- ננסה להוכיח תכונה (אינואריאנטה, משתמר) של תוכנית פשוטה. ערך המשתנה `counter` שווה למספר הקריאות לשרות `m()`:

```java
/** @inv counter == #calls for m() */
public class StaticMemberExample {

   public static int counter; //initialized by default to 0

   public static void m() {
      counter++;
   }
}
```

- נוכיח זאת **באינדוקציה** על מספר הקריאות ל- **`m()`**, עבור כל קטע קוד שיש בו התייחסות למחלקה **`StaticMemberExample`**

"הוכחה"

- **<u>מקרה בסיס (n=0):</u>** אם בקטע קוד מסוים אין קריאה למתודה `m()` אזי בזמן טעינת המחלקה `StaticMemberExample` לזיכרון התוכנית מאותחל המשתנה `counter` לאפס. והדרוש נובע.

- **<u>הנחת האינדוקציה (n=k):</u>** נניח כי קיים k טבעי כלשהו כך שבסופו של כל קטע קוד שבו k קריאות לשרות `m()` ערכו של counter הוא k.

- **<u>צעד האינדוקציה (n=k+1):</u>** נוכיח כי בסופו של קטע קוד עם k+1 קריאות ל `m()` ערכו של counter הוא k+1

<u>הוכחה</u>: יהי קטע הקוד שבו k+1 קריאות ל `m()`. נתבונן בקריאה האחרונה ל- `m()`. קטע הקוד עד לקריאה זו הוא קטע עם k קריאות בלבד. ולכן לפי הנחת האינדוקציה בנקודה זו `counter==k`. בעת ביצוע המתודה `m()` מתבצע `counter++` ולכן ערכו עולה ל k+1. מכיוון שזוהי הקריאה האחרונה ל `m()` בתוכנית, ערכו של `counter` עד לסוף התוכנית ישאר k+1 כנדרש. **מ.ש.ל.**

דוגמא נגדית

```
public class CounterExample {

   public static void main(String[] args) {
        StaticMemberExample.m();
        StaticMemberExample.m();
        StaticMemberExample.counter++;
   }
}
```

- מה היה חסר ב"הוכחה" בשקף הקודם?
- לא לקחנו בחשבון שניתן לשנות את **counter** גם מחוץ למחלקה שבה הוגדר
- כלומר, נכונות הטענה תלויה **באופן השימוש** של הלקוחות בקוד
- לצורך שמירה על הנכונות יש צורך **למנוע** מלקוחות המחלקה את הגישה למשתנה **counter**

נראות פרטית (private visibility)

- הגדרת משתנה או שרות כ **private** מאפשרים גישה אליו רק מתוך המחלקה שבה הוגדר:

```
/** @inv counter == #calls for m() */
public class StaticMemberExample {

    private static int counter; //initialized by default to 0

    public static void m() {
        counter++;
    }
}
```

```
public class CounterExample {

    public static void main(String[] args) {
        StaticMemberExample.m();
        StaticMemberExample.m();
        StaticMemberExample.counter++;
        System.out.println("main(): m() was called " +
                           StaticMemberExample.counter + " times");
    }
}
```

הסתרת מידע והכמסה

- שימוש ב- `private` "תוחם את הבאג" ונאכף על ידי המהדר
- כעת אם קיימת שגיאה בניהול המשתנה **`counter`** היא לבטח נמצאת בתוך המחלקה **`StaticMemberExample`** ואין צורך לחפש אותה בקרב הלקוחות (שעשויים להיות רבים)
- תיחום זה מכונה **הכמסה** (encapsulation)
- את ההכמסה הישגנו בעזרת **הסתרת מידע** (information hiding) מהלקוח
- בעיה – ההסתרה גורפת מדי - כעת הלקוח גם לא יכול **לקרוא** את ערכו של **`counter`**

גישה מבוקרת

- נגדיר מתודות גישה ציבוריות (public) אשר יחזירו את ערכו של המשתנה הפרטי

```
/** @inv getCounter() == #calls for m() */
public class StaticMemberExample {

   private static int counter;

   public static int getCounter() {
       return counter;
   }

   public static void m() {
       counter++;
   }
}
```

המשתמר הוא חלק מהחוזה של הספק כלפי הלקוח ולכן הוא מנוסח בשפה שהלקוח מבין

גישה מבוקרת

- הלקוחות ניגשים למונה דרך המתודה שמספק להם הספק

```
public class CounterExample {

   public static void main(String[] args) {
      StaticMemberExample.m();
      StaticMemberExample.m();
      // StaticMemberExample.counter++; - access forbidden

      System.out.println("main(): m() was called " +
                StaticMemberExample.getCounter() + " times");
   }
}
```

משתמר הייצוג

- ראינו שימוש בחוזה של מחלקה כדי לבטא בצורה מפורשת את גבולות האחריות עם לקוחות המחלקה

- אולם, ניתן להשתמש במתודולוגיה של "עיצוב ע"פ חוזה" גם "לצורכי פנים"

- כשם שהחוזה מבטא הנחות והתנהגות בצורה פורמלית יותר מאשר הערות בשפה טבעית, כך ניתן להוסיף טענות בולאניות לגבי היבטים של המימוש

- כדי שלא לבלבל את הלקוחות עם משתמר המכיל ביטויים שאינם מוכרים להם, נגדיר **משתמר ייצוג** המיועד לספקי המחלקה בלבד

משתמר הייצוג

- משתמר ייצוג (representation invariant, Implementation invariant) הוא בעצם משתמר המכיל מידע פרטי (private)
- לדוגמא:

```
/** @inv getCounter() == #calls for m()
 *  @imp_inv counter == #calls for m()
 */
public class StaticMemberExample {

   private static int counter;

   public static int getCounter() {
      return counter;
   }
}
```

תנאי בתר ייצוגי

- גם בתנאי בתר עלולים להיות ביטויים פרטיים שנרצה להסתיר מהלקוח:

```
/** @imp_post isIntialized */
public static void init(String login, String password)
```

- אבל לא בתנאי קדם של מתודות ציבוריות
- מדוע?

מתודות עזר

- ניתן למנוע גישה **לשרות** ע"י הגדרתו כ `private`
- הדבר מאפיין שרותי עזר, אשר אין רצון לספק לחשוף אותם כלפי חוץ
- סיבות אפשריות להגדרת שרותים כפרטיים:
  - השרות מפר את המשתמר ויש צורך לתקנו אחר כך
  - השרות מבצע חלק ממשימה מורכבת, ויש לו הגיון רק במסגרתה (לדוגמא שרות שנוצר ע"י חילוץ קטע קוד למתודה, extract method)
  - הספק מעוניין לייצא מספר שרותים מצומצם, וניתן לבצע את השרות הפרטי בדרך אחרת
  - השרות מפר את רמת ההפשטה של המחלקה (לדוגמא sort המשתמש ב quicksort כמתודת עזר)

נראות ברמת החבילה
(package friendly)

- כאשר איננו מציינים הרשאת גישה (נראות) של תכונה או מאפיין קיימת ברירת מחדל של **נראות ברמת החבילה**
- כלומר ניתן לגשת לתכונה (משתנה או שרות) אך ורק מתוך מחלקות שבאותה החבילה (package) כמו המחלקה שהגדירה את התכונה
- ההיגיון בהגדרת נראות כזו, הוא שמחלקות באותה החבילה כנראה נכתבות באותו ארגון (אותו צוות בחברה) ולכן הסיכוי שיכבדו את המשתמרים זו של זו גבוה
- נראות ברמת החבילה היא יצור כלאיים לא שימושי:
  - מתירני מדי מכדי לאכוף את המשתמר
  - קפדני מדי מכדי לאפשר גישה חופשית

הוכחת החוזה

- נוסף על הוכחת נכונות המשתמר, נרצה להוכיח כי החוזה של כל אחת מהמתודות מתקיים
  - כלומר בהינתן שתנאי הקדם מתקיים נובע תנאי האחר
- מבנה הוכחות אלו כולל בדיקת כל המקרים האפשריים או הוכחה באינדוקציה (בדומה למה שראינו בהוכחת המשתמר)
  - אנו **מניחים** כי תנאי הקדם מתקיים בכניסה לשרות **ומוכיחים** כי תנאי האחר מתקיים ביציאה מהשרות
- להוכחות כאלו יש חשיבות בבניית אמינות לספריות תוכנה, בפרט אם הם משמשות במערכות חיוניות
- דוגמאות לכך ניתן למצוא בקובץ הדוגמאות באתר הקורס – "הוכחת נכונות של שרותים"

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 כטיפוסים 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 ב-2 מחלקות: מחרוזת ומערך

- ראינו כי עבודה עם מחלקות מערבת שתי ישויות נפרדות:
  - **העצם**: המכיל את המידע
  - **ההפנייה**: משתנה שדרכו ניתן לגשת לעצם

- זאת בשונה ממשתנים יסודיים (טיפוסים פרימיטיביים)

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 ייווצרו 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 **day**, **month** ו- **year** 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 בזכרון נמצאים השדות **day**, **month** ו- **year**?
- תשובה: הם עוד לא נמצאים! הם יווצרו 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?
- ראינו שמחלקה 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

```java
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 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 **פונקצית 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 השיעור)

```java
public class MyDate {

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

    // ...
}
```

הגדרת בנאי ל MyDate

```java
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

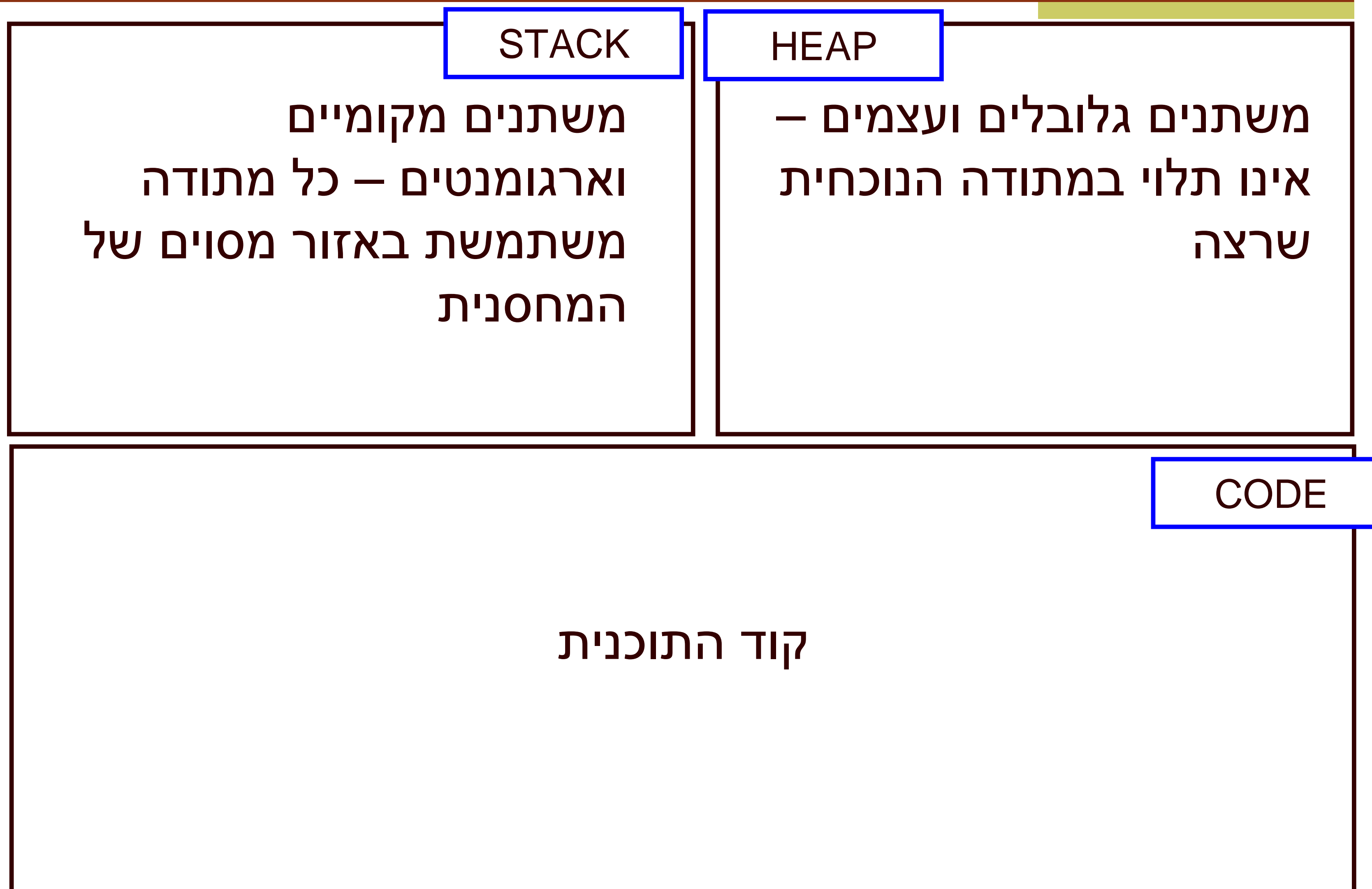
- בדוגמא הבאה נראה כיצד מייצר הקומפיילר עבורנו את ההפניה **this** עבור כל בנאי וכל שרות מופע
- נתבונן במחלקה **Point** המייצגת נקודה במישור הדו מימדי. כמו כן המחלקה מנהלת מעקב בעזרת משתנה גלובלי (סטטי) אחר מספר העצמים שנוצרו מהמחלקה
- בהמשך הקורס נציג מימוש מלא ומעניין יותר של המחלקה, אולם כעת לצורך פשטות הדוגמא נסתפק בבנאי, שדה מחלקה, 2 שדות מופע ו-3 שרותי מופע

```java
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 preconditon - for responsible clients
       * @pre  (newX > 0.0 && newX < 100.0)
       * @post getX() == newX
       */
      public void doSetX(double newX) {
         x = newX;
      }

      // More methods...
}
```

PointUser

```java
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());
    }

}
```

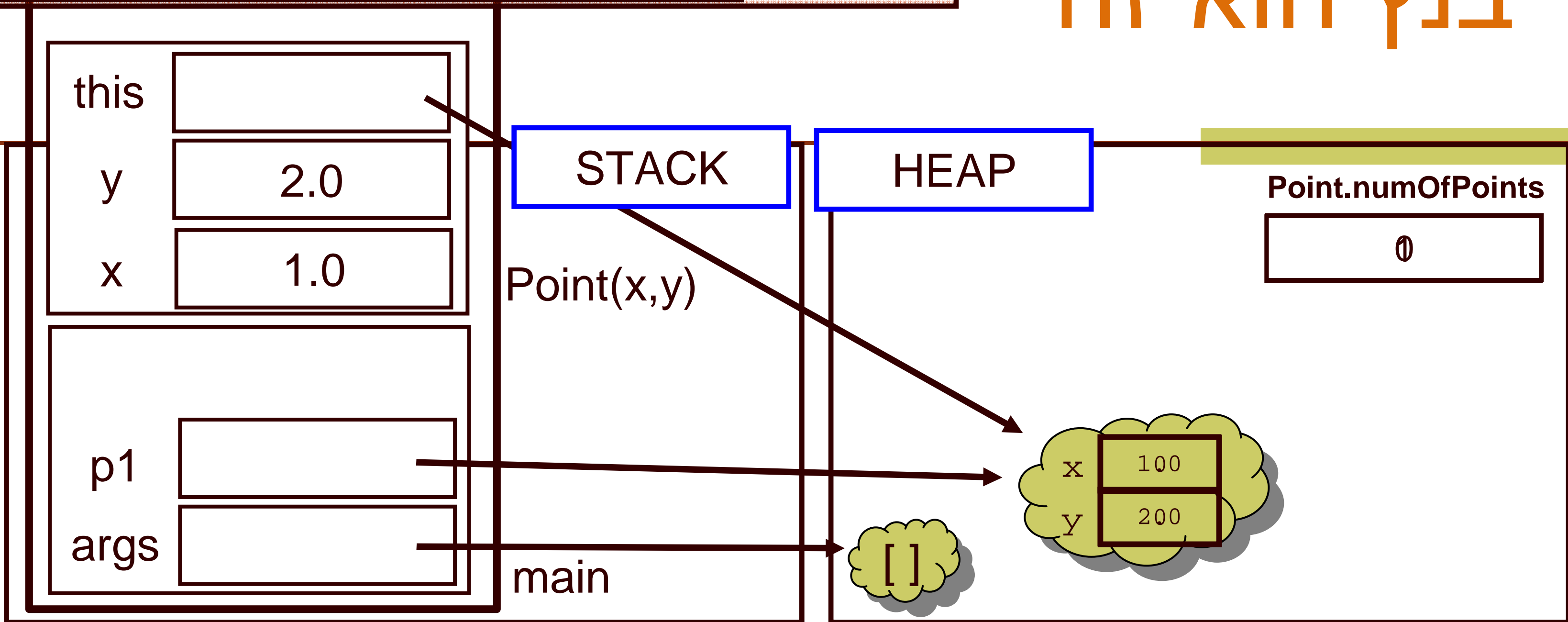
מודל הזיכרון של Java



"בנץ הוא זה"

במהלך ריצת בנאי, ההפניה this מצביעה
על העצם שזה עתה הוקצה

בכל הפעלה של שרות מופעל, ההפניה this מצביעה לעצם המטרה (target)
שעליו הופעל השרות לעצם זה

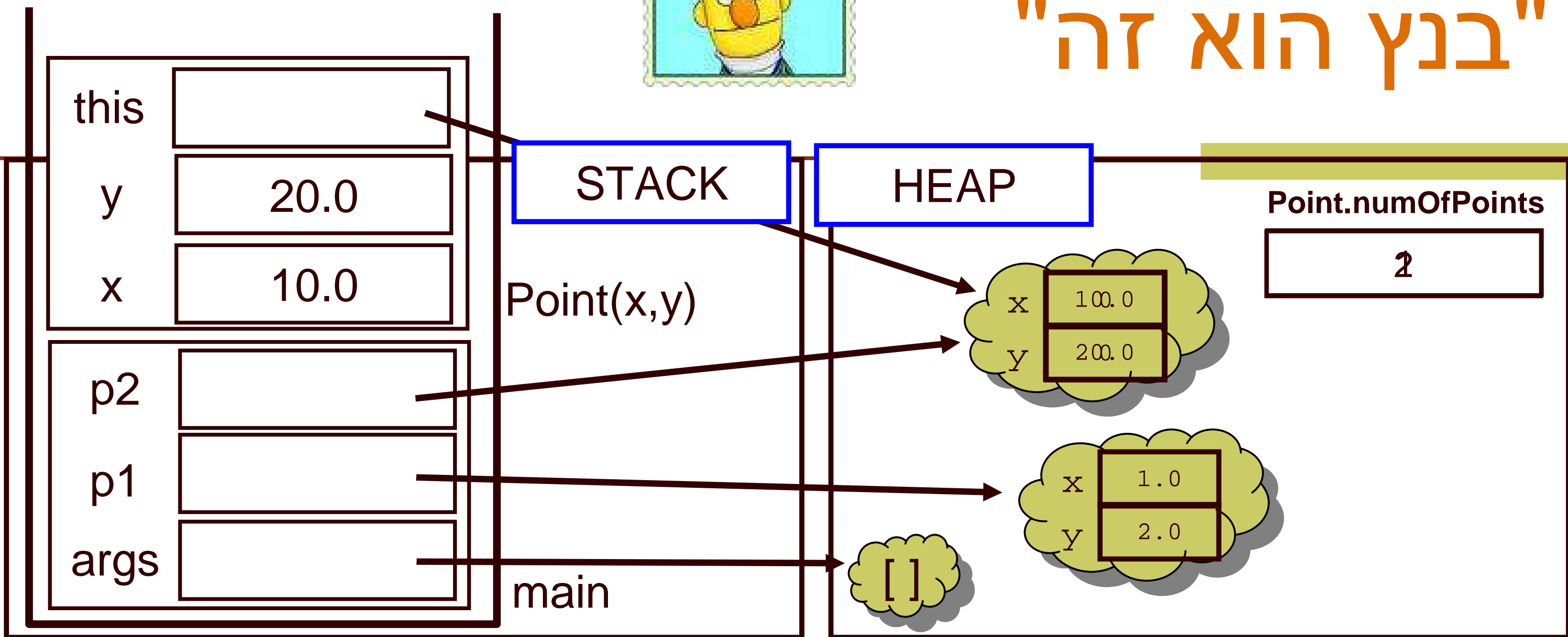


CODE

```java
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());
    }

}
```

```java
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; }
```

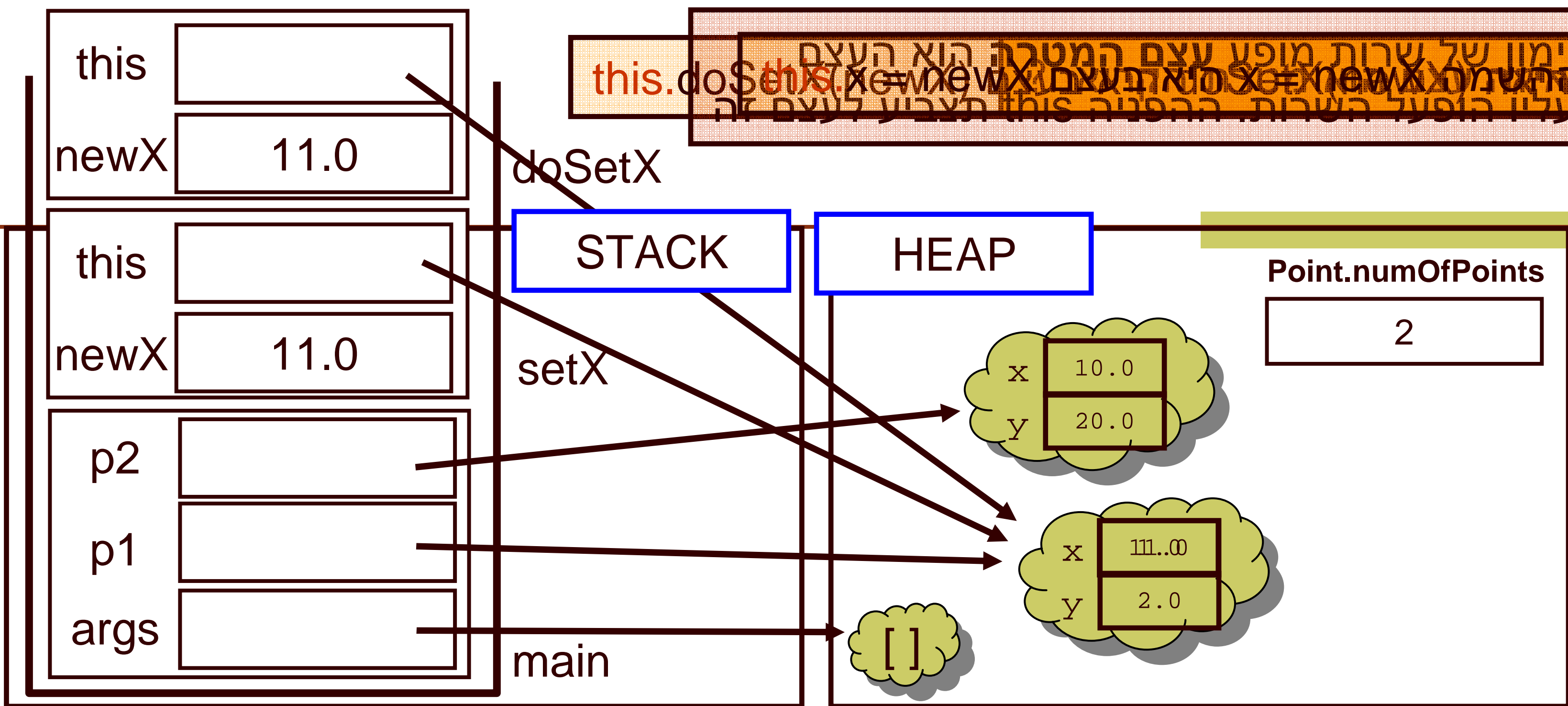
"בנץ הוא זה"




```java
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());
    }
}
```

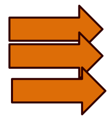
```java
public class Point {
    public Point(double x, double y){
        this.x = x;
        this.y = y;
        numOfPoints++;
    }

    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; }
```

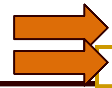
CODE



```java
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());
    }

}
```

```java
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)
            this.doSetX(newX);
    }

    public void doSetX(double newX)
    { this.x = newX; }
```





CODE

this.x = newX

this	
newX	21.0

doSetX

this	
newX	21.0

setX

p2	
p1	
args	

main

STACK

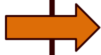
HEAP

Point.numOfPoints: 2

Point object: x = 21.0, y = 20.0

Point object: x = 11.0, y = 2.0

[ ]

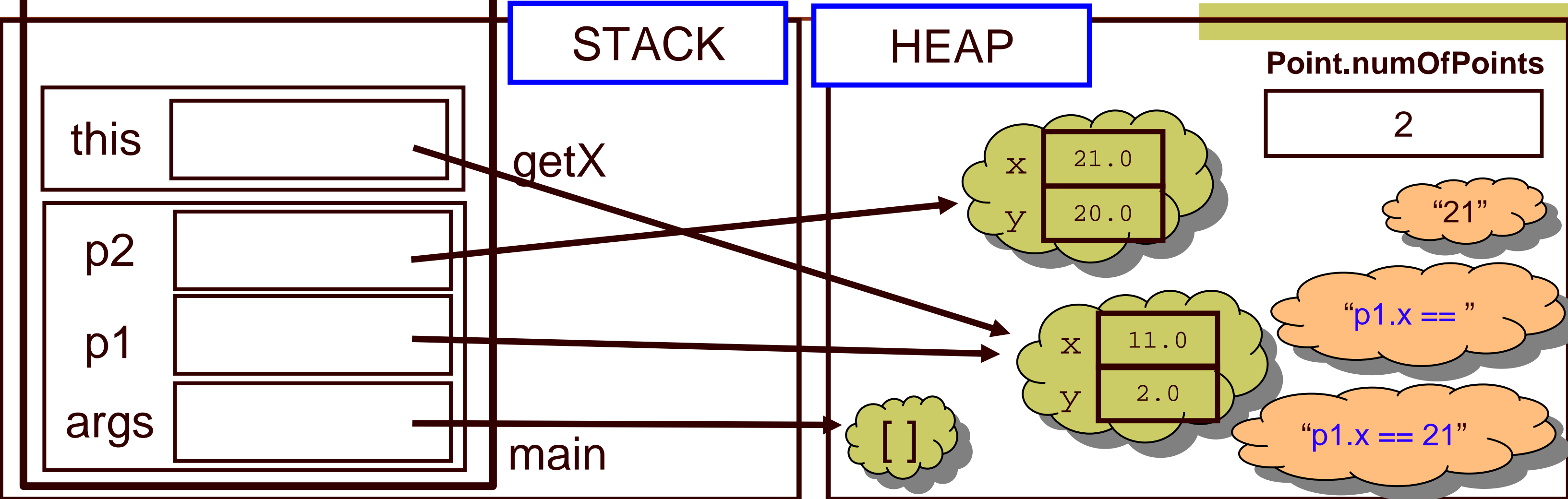


```
public class PointUser {

    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());
    }

}
```

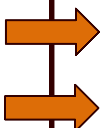
```
public class Point {

    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)
            this.doSetX(newX);
    }

    public void doSetX(double newX)
    { this.x = newX; }
```

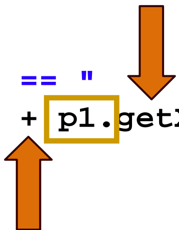
CODE

המשפט "return x" הוא בעצם "return this.x"




```
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 this.x; }

    public void setX(double newX) {
        if(newX > 0.0 && newX < 100.0)
            doSetX(newX);
    }

    public void doSetX(double newX)
    { this.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 גלובלים. קיים עותק אחד שלהם, שנוצר בעת טעינת קוד המחלקה לזכרון, ללא קשר ליצירת עצמים מאותה המחלקה