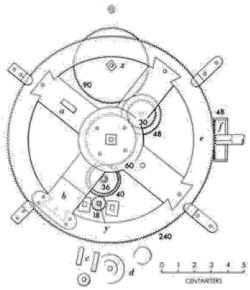
תוכנה 1 בשפת Java

שיעור מספר 3: "מחלקות וטיפוסים"

בית הספר למדעי המחשב

אוניברסיטת תל אביב

על סדר היום



- משתני מחלקה – המשך משבוע שעבר
- מודל הזיכרון של Java
  - Heap and Stack
  - העברת ארגומנטים
- מנגנוני שפת Java – בתרגול עצמי
- עצמים ושירותי מופע

משתני מחלקה

- עד כה ראינו משתנים מקומיים – משתנים זמניים המוגדרים בתוך מתודה, בכל קריאה למתודה הם נוצרים וביציאה ממנה הם נהרסים

- ב Java ניתן גם להגדיר **משתנים גלובליים** (global variables)
  - מכונים שדות סטטיים (static fields/members)

- משתנים אלו יוגדרו בתוך גוף המחלקה אך מחוץ לגוף של מתודה כלשהי, ויסומנו ע"י המציין `static`.
  - יכולה להיות ניראות שונה (public/private)

משתני מחלקה לעומת משתנים מקומיים

- משתנים אלו, שונים ממשתנים מקומיים בכמה מאפיינים:
  - **תחום הכרות:** כתלות בנראות (נראה נראויות שונות בהמשך הקורס), מוכרים בכל הקוד, ולא רק בתוך פונקציה מסויימת.
  - **משך קיום:** אותו עותק של משתנה נוצר בזמן טעינת הקוד לזיכרון ונשאר קיים בזיכרון התוכנית כל עוד המחלקה בשימוש
  - **אתחול:** משתנים סטטיים מאותחלים בעת יצירתם. אם המתכנתת לא הגדירה להם ערך אתחול - יאותחלו לערך ברירת המחדל לפי טיפוסם (`0, false, null`)
  - **הקצאת זיכרון:** הזיכרון המוקצה להם נמצא באזור ה Heap (ולא באזור ה- Stack)

נשתמש במשתנה גלובלי counter כדי לספור את מספר הקריאות למתודה m():

```
public class StaticMemberExample {

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

   public static void m() {
      int local = 0;
      counter++;
      local++;
      System.out.println("m(): local is " + local +
                         "\tcounter is " + counter);
   }

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

שם מלא

- ניתן לפנות למשתנה counter גם מתוך קוד במחלקה אחרת, אולם יש צורך לציין את שמו המלא (qualified name)
- במחלקה שבה הוגדר משתנה גלובלי ניתן לגשת אליו תוך ציון שמו המלא או שם המזהה בלבד (unqualified name)
- בדומה לצורת הקריאה לשרותי מחלקה

```
public class AnotherClass {

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

זה סופי

- ניתן לקבע ערך של משתנה ע"י ציון המשתנה כ **`final`**
- למשתנה שהוא **`final`** ניתן לבצע השמה פעם אחת בדיוק.
  כל השמה נוספת לאותו משתנה תגרור שגיאת קומפילציה
- דוגמא:

```
public final static long uniqueID = ++counter;
```

- מוסכמה מקובלת היא שמות משתנים המציינים קבועים ב-UPPERCASE כגון:

```
public final static double FOOT = 0.3048;
public final static double PI = 3.14159265358979323384;
```

העברת ארגומנטים

- כאשר מתבצעת קריאה לשרות, ערכי הארגומנטים נקשרים לפרמטרים הפורמליים של השרות לפי הסדר, ומתבצעת השמה לפני ביצוע גוף השרות.
- בהעברת ערך לשרות הערך **מועתק** לפרמטר הפורמלי
- צורה זאת של העברת פרמטרים נקראת **call by value**
- כאשר הארגומנט המועבר הוא **הפנייה** (התייחסות, reference) העברת הפרמטר **מעתיקה את ההתייחסות.**
  - בשפות תכנות אחרות ניתן לבצע העברה של המצביע עצמו

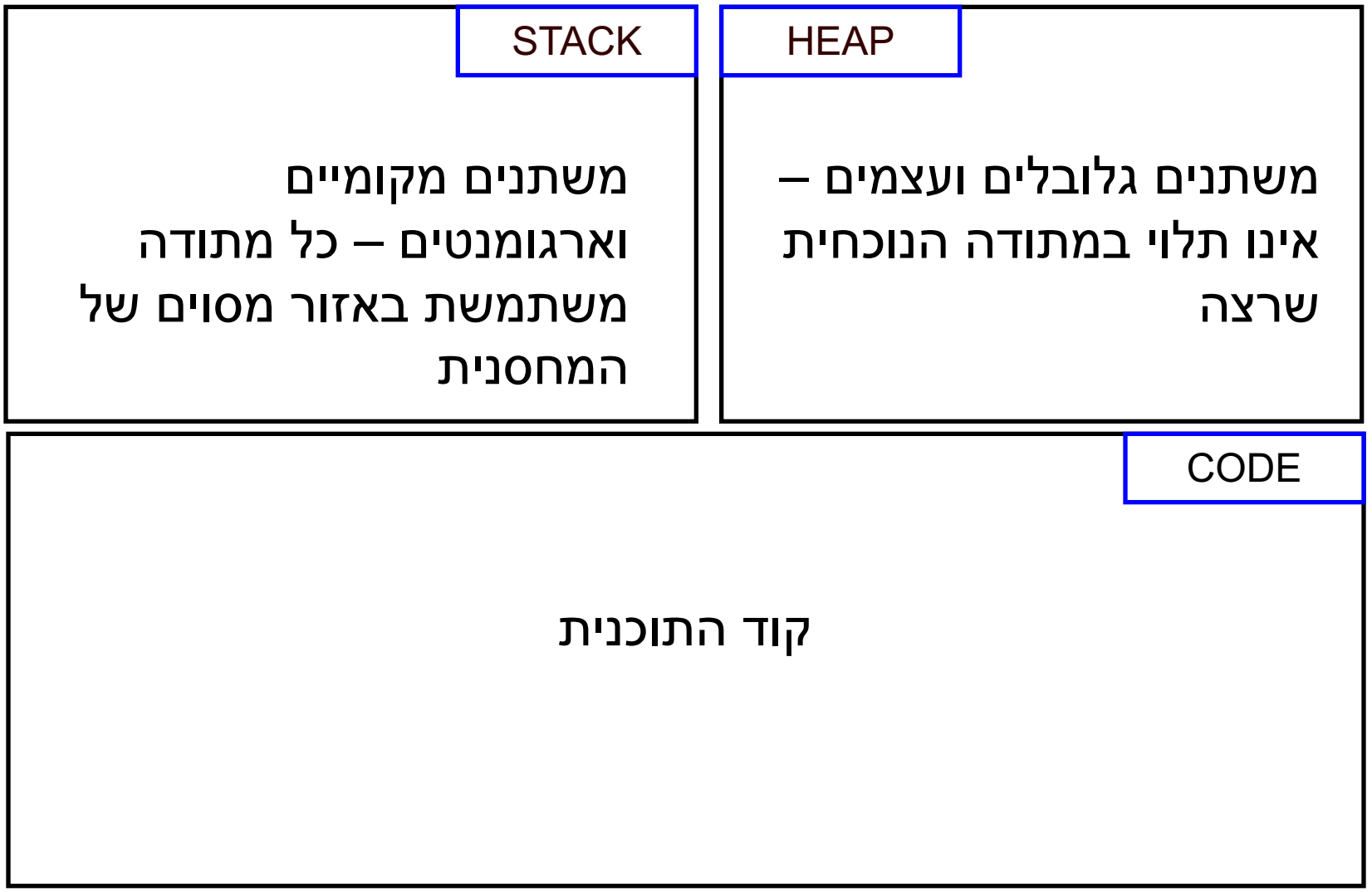
ב Java גם reference מועבר by value

העברת פרמטרים by value

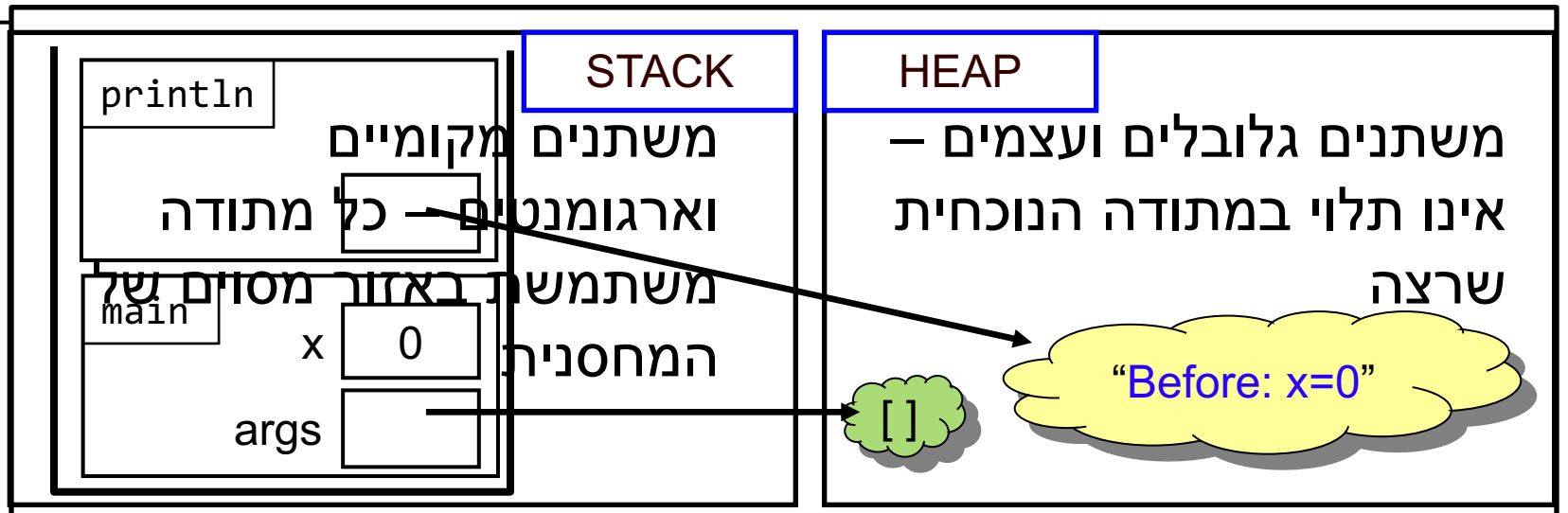
- העברת פרמטרים by value (ע"י העתקה) יוצרת מספר מקרים מבלבלים, שידרשו מאיתנו הכרות מעמיקה יותר עם מודל הזיכרון של Java
- למשל, מה מדפיס הקוד הבא?

```
public class CallByValue {

   public static void setToFive(int arg){
      arg = 5;
   }

   public static void main(String[] args) {
      int x = 0;
      System.out.println("Before: x=" + x);
      setToFive(x);
      System.out.println("After: x=" + x);
   }
}
```

מודל הזיכרון של Java



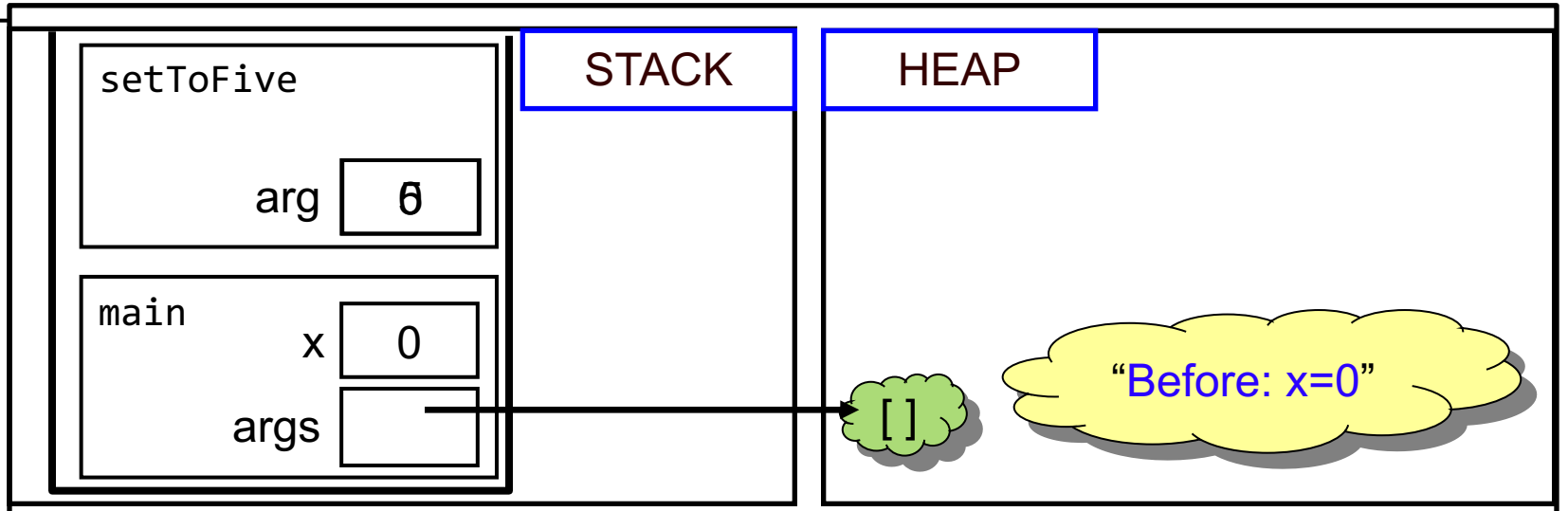
Primitives by value




```
public class CallByValue {

    public static void setToFive(int arg){
       arg = 5;
    }

    public static void main(String[] args) {
       int x = 0;
       System.out.println("Before: x=" + x);
       setToFive(x);
       System.out.println("After: x=" + x);
    }
}
```

CODE

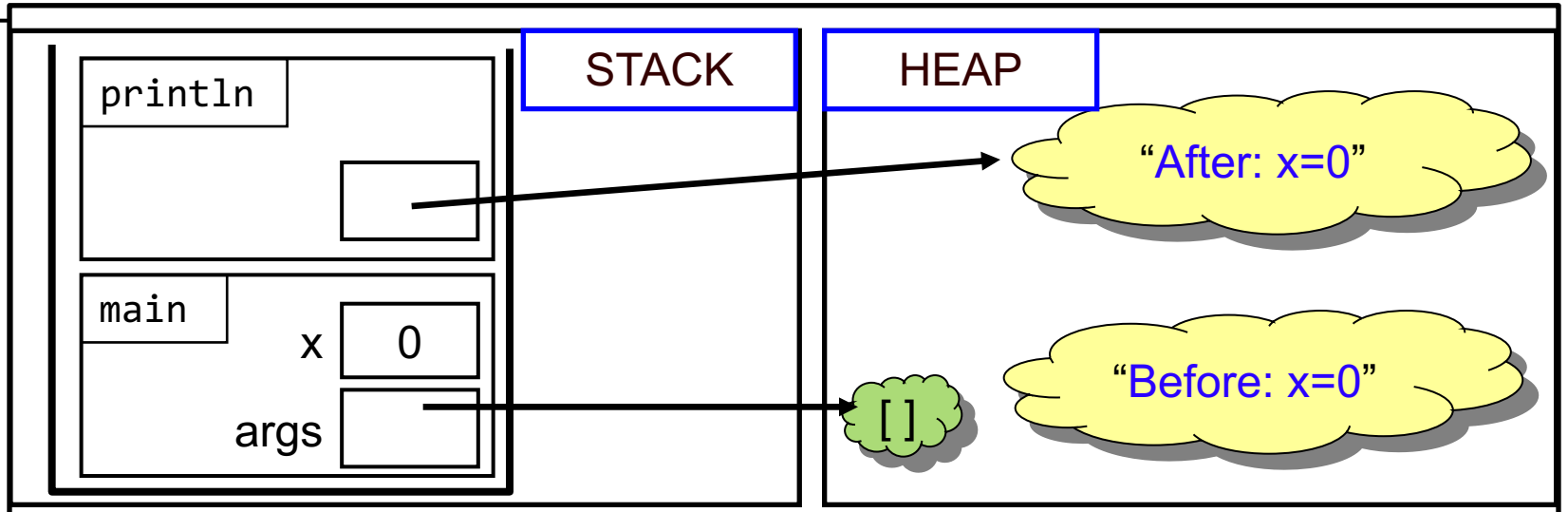
Primitives by value




```
public class CallByValue {

    public static void setToFive(int arg){
        arg = 5;
    }

    public static void main(String[] args) {
        int x = 0;
        System.out.println("Before: x=" + x);
        setToFive(x);
        System.out.println("After: x=" + x);
    }
}
```

CODE

Primitives by value



CODE

```
public class CallByValue {

    public static void setToFive(int arg){
        arg = 5;
    }

    public static void main(String[] args) {
        int x = 0;
        System.out.println("Before: x=" + x);
        setToFive(x);
        System.out.println("After: x=" + x);
    }
}
```

לאחר ש main מסיים את
איזורי הזיכרון שהוקצה
עבורה על ה- Stack משוחרר

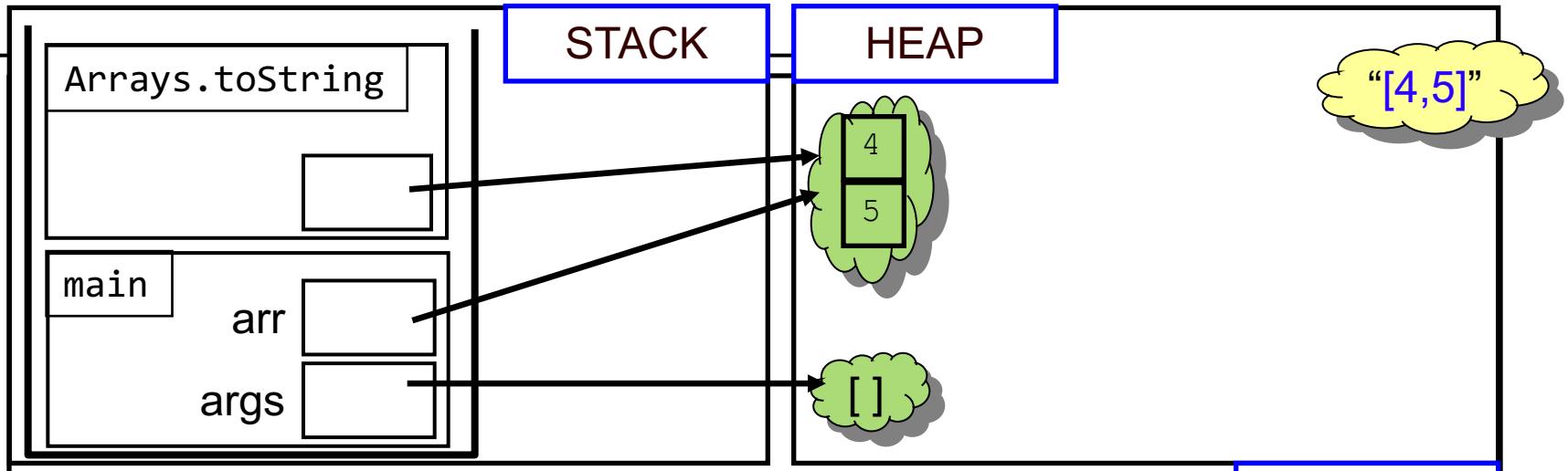
שמות מקומיים

- בדוגמא ראינו כי הפרמטר הפורמלי **`arg`** קיבל את הערך של הארגומנט x
- בחירת השמות השונים אינה משמעותית - יכולנו לקרוא לשני המשתנים באותו שם ולקבל התנהגות זהה
- שם של משתנה מקומי **מסתיר** משתנים בשם זהה הנמצאים בתחום עוטף או גלובלים
  - נראה את ההתנהגות הזו בהמשך השיעור, בדוגמא של *בנאים*.
- מתודה מכירה רק משתני מחסנית הנמצאים באזור שהוקצה לה על המחסנית (frame)

- מה יקרה אם המשתנה המקומי x שהועבר היה מטיפוס הפנייה? למשל, מה מדפיס הקוד הבא?

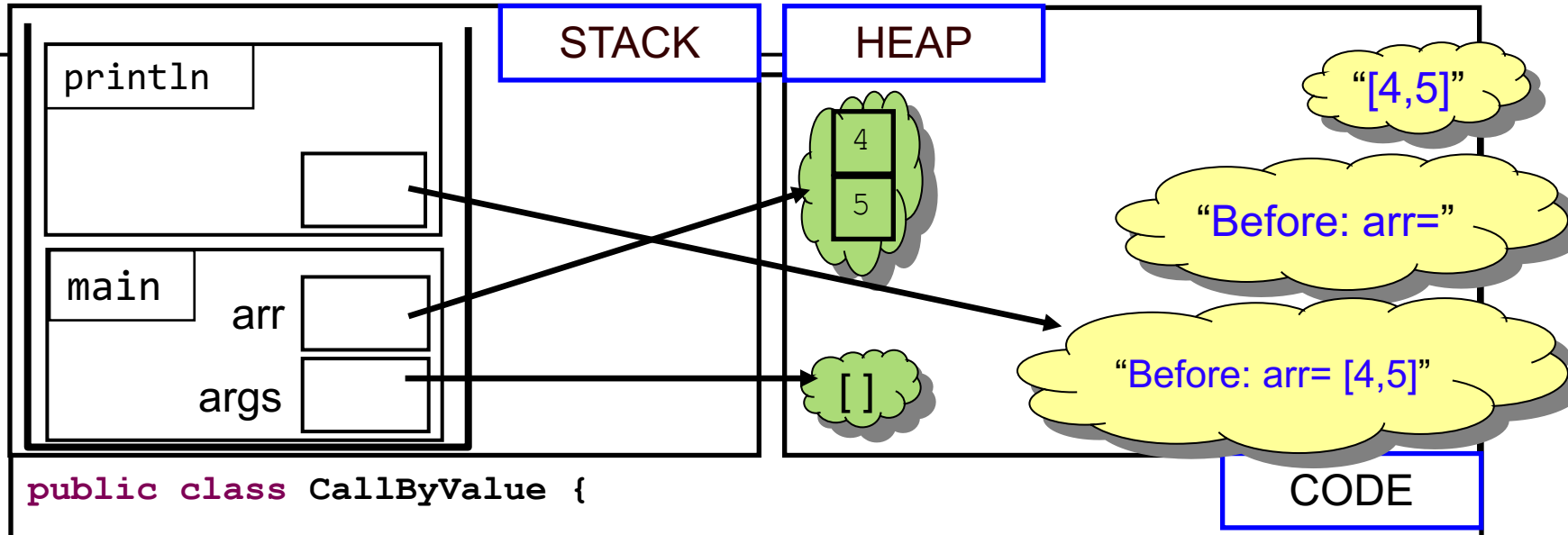
```
import java.util.Arrays; //explained later...

public class CallByValue {

    public static void setToZero(int [] arr){
        arr = new int[3];
    }

    public static void main(String[] args) {
        int [] arr = {4,5};
        System.out.println("Before: arr=" + Arrays.toString(arr));
        setToZero(arr);
        System.out.println("After: arr=" + Arrays.toString(arr));
    }
}
```

Reference by value



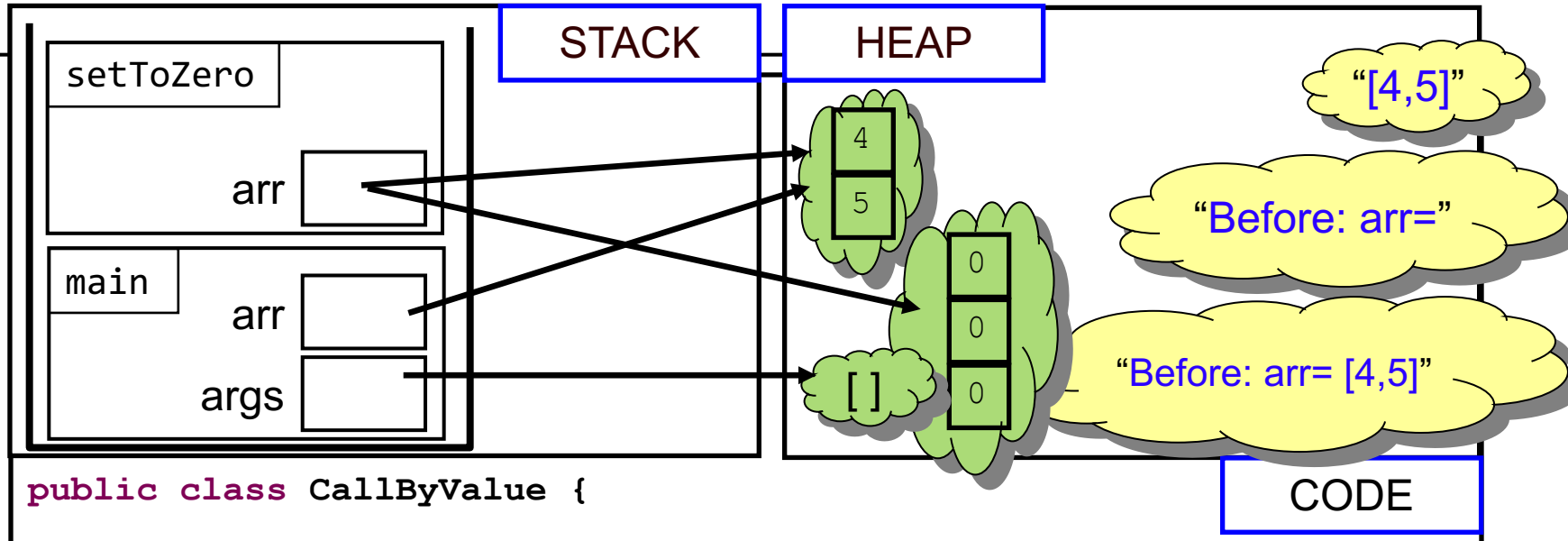

```
public class CallByValue {

    public static void setToZero(int [] arr){
        arr = new int[3];
    }

    public static void main(String[] args) {
        int [] arr = {4,5};
        System.out.println("Before: arr=" + Arrays.toString(arr));
        setToZero(arr);
        System.out.println("After: arr=" + Arrays.toString(arr));
    }
}
```

Reference by value



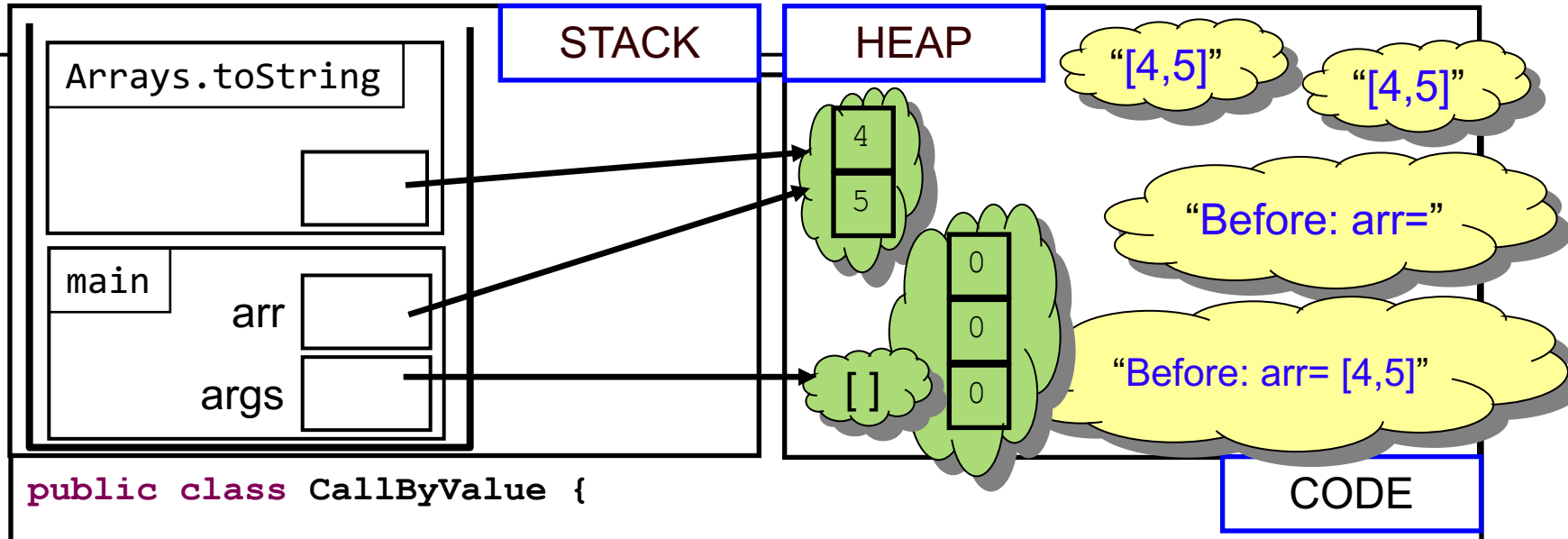

```
public class CallByValue {

    public static void setToZero(int [] arr){
        arr = new int[3];
    }

    public static void main(String[] args) {
        int [] arr = {4,5};
        System.out.println("Before: arr=" + Arrays.toString(arr));
        setToZero(arr);
        System.out.println("After: arr=" + Arrays.toString(arr));
    }
}
```

Reference by value



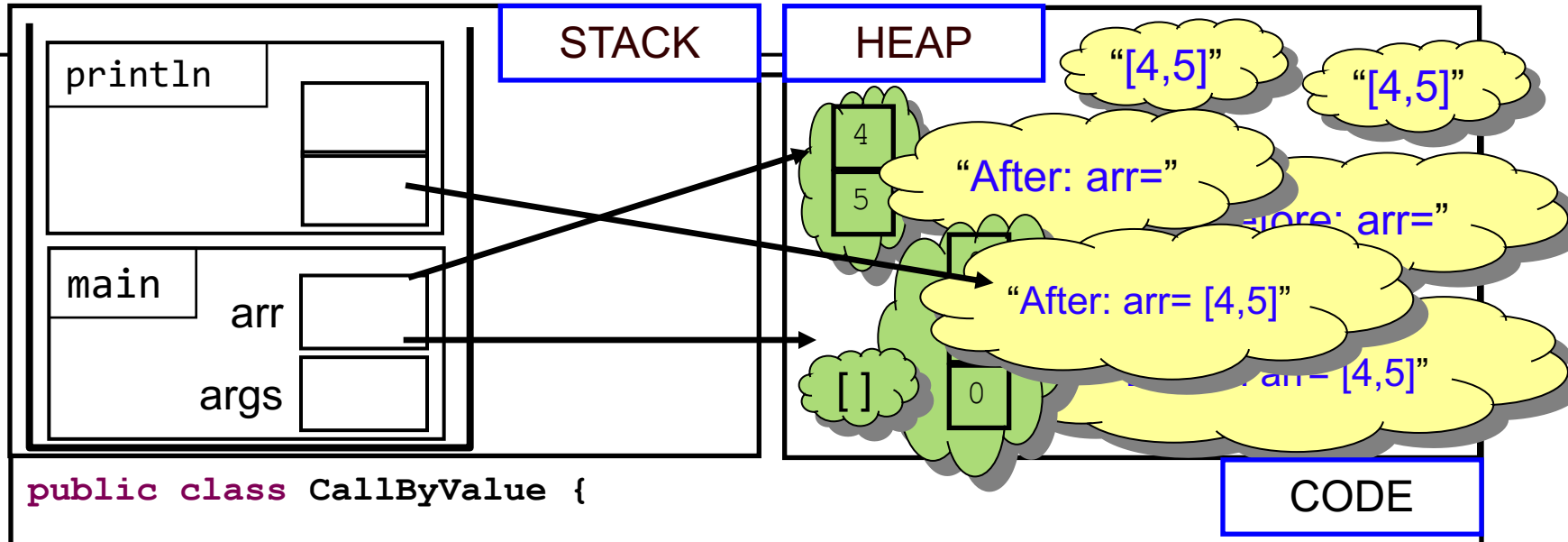

```
public class CallByValue {

    public static void setToZero(int [] arr){
        arr = new int[3];
    }

    public static void main(String[] args) {
        int [] arr = {4,5};
        System.out.println("Before: arr=" + Arrays.toString(arr));
        setToZero(arr);
        System.out.println("After: arr=" + Arrays.toString(arr));
    }
}
```

Reference by value




```
public class CallByValue {

    public static void setToZero(int [] arr){
        arr = new int[3];
    }

    public static void main(String[] args) {
        int [] arr = {4,5};
        System.out.println("Before: arr=" + Arrays.toString(arr));
        setToZero(arr);
        System.out.println("After: arr=" + Arrays.toString(arr));
    }
}
```

Reference by value



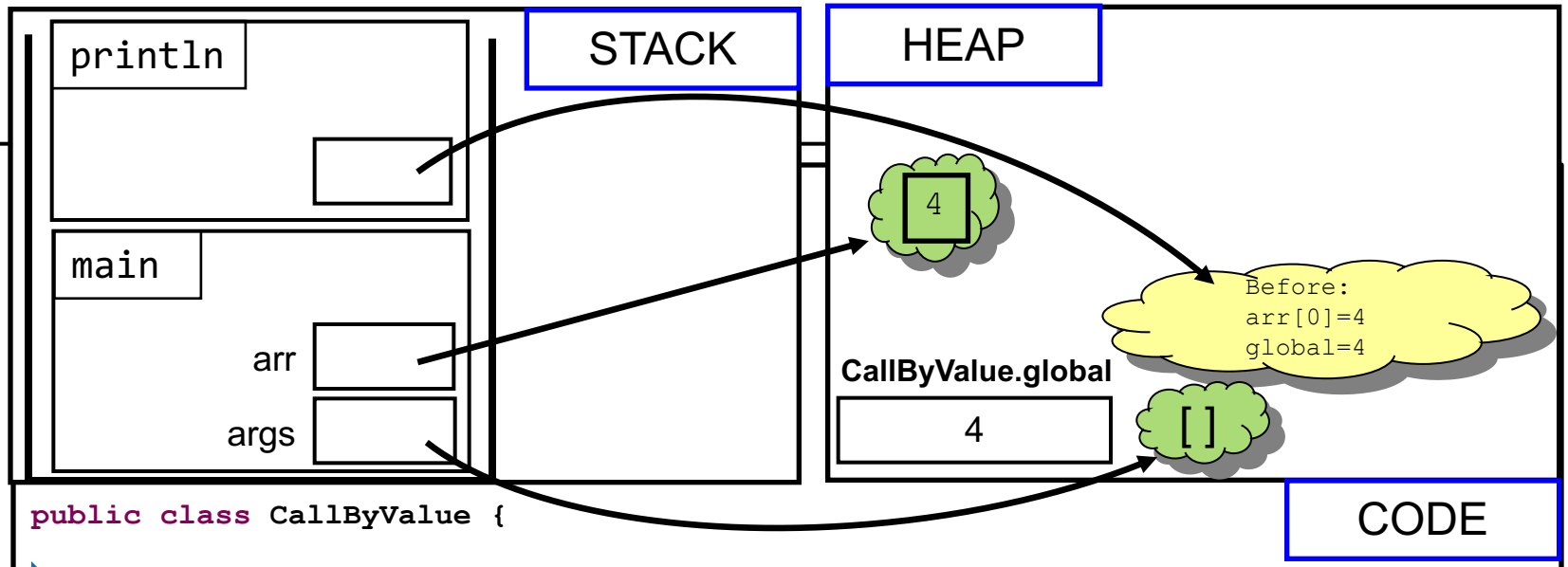

```
public class CallByValue {

    public static void setToZero(int [] arr){
        arr = new int[3];
    }

    public static void main(String[] args) {
        int [] arr = {4,5};
        System.out.println("Before: arr=" + Arrays.toString(arr));
        setToZero(arr);
        System.out.println("After: arr=" + Arrays.toString(arr));
    }
}
```

הפונקציה הנקראת והעולם שבחוץ

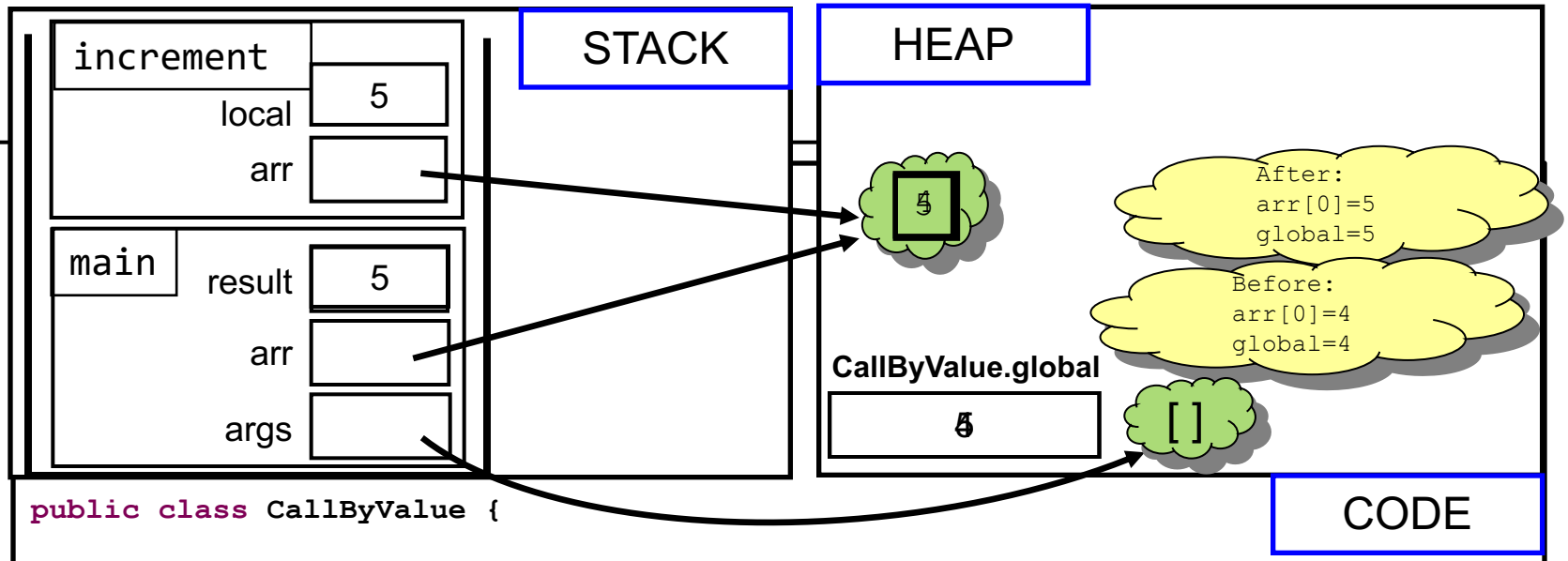
- בשיטת העברה by value לא יעזור למתודה לשנות את הארגומנט שקיבלה, מכיוון שהיא מקבלת עותק
- אז איך יכולה מתודה להשפיע על ערכים במתודה שקראה לה?
  - ע"י ערך מוחזר
  - ע"י גישה למשתנים או עצמים שהוקצו ב- Heap
- מתודות שמשנות את תמונת הזיכרון נקראות בהקשרים מסוימים Mutators או Transformers

מה מדפיסה התוכנית הבאה?

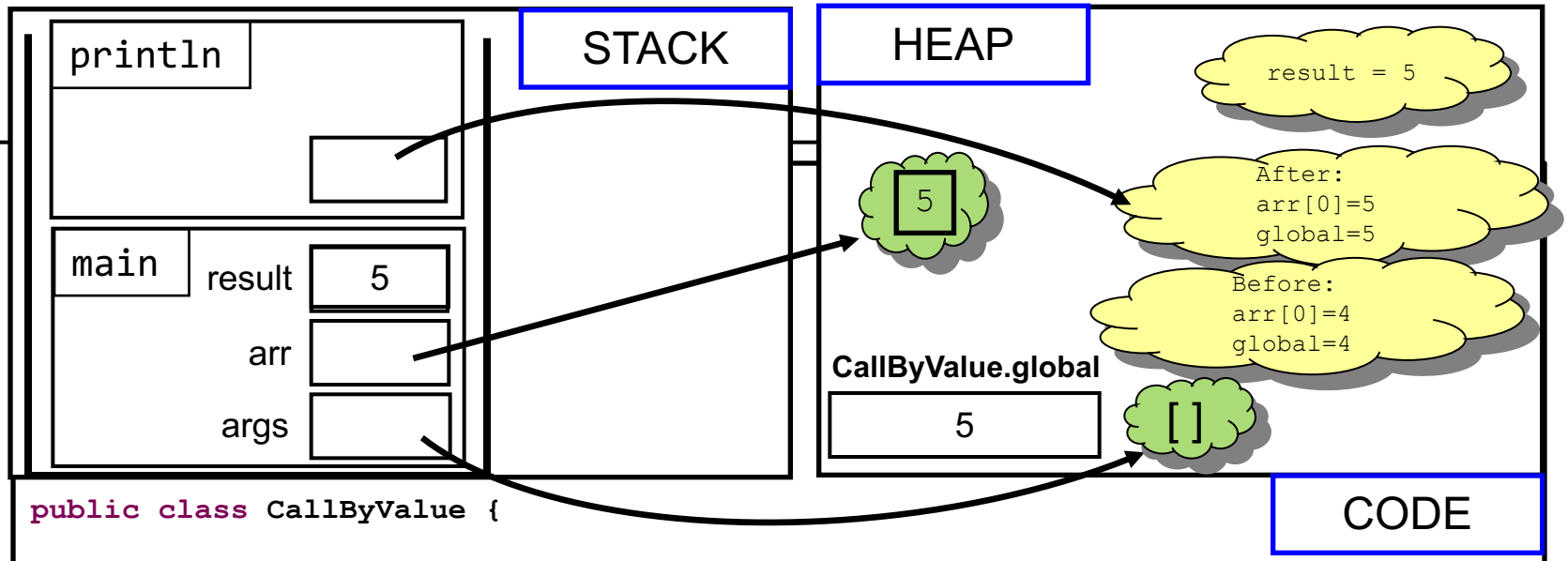
```
public class CallByValue {

    static int global = 4;

    public static int increment(int [] arr){
        int local = 5;
        arr[0]++;
        global++;
        return local;
    }

    public static void main(String[] args) {
        int [] arr = {4};
        System.out.println("Before:\narr[0]=" + arr[0] +
                                    "\nglobal=" + global);
        int result = increment(arr);
        System.out.println("After:\narr[0]=" + arr[0] +
                                    "\nglobal=" + global);
        System.out.println("result = " + result);
    }
}
```




```
public class CallByValue {

    static int global = 4;

    public static int increment(int [] arr){
        int local = 5;
        arr[0]++;
        global++;
        return local;
    }

    public static void main(String[] args) {
        int [] arr = {4};
        System.out.println("Before:\narr[0]=" + arr[0] + "\nglobal=" + global);
        int result = increment(arr);
        System.out.println("After:\narr[0]=" + arr[0] + "\nglobal=" + global);
        System.out.println("result = " + result);
    }
}
```

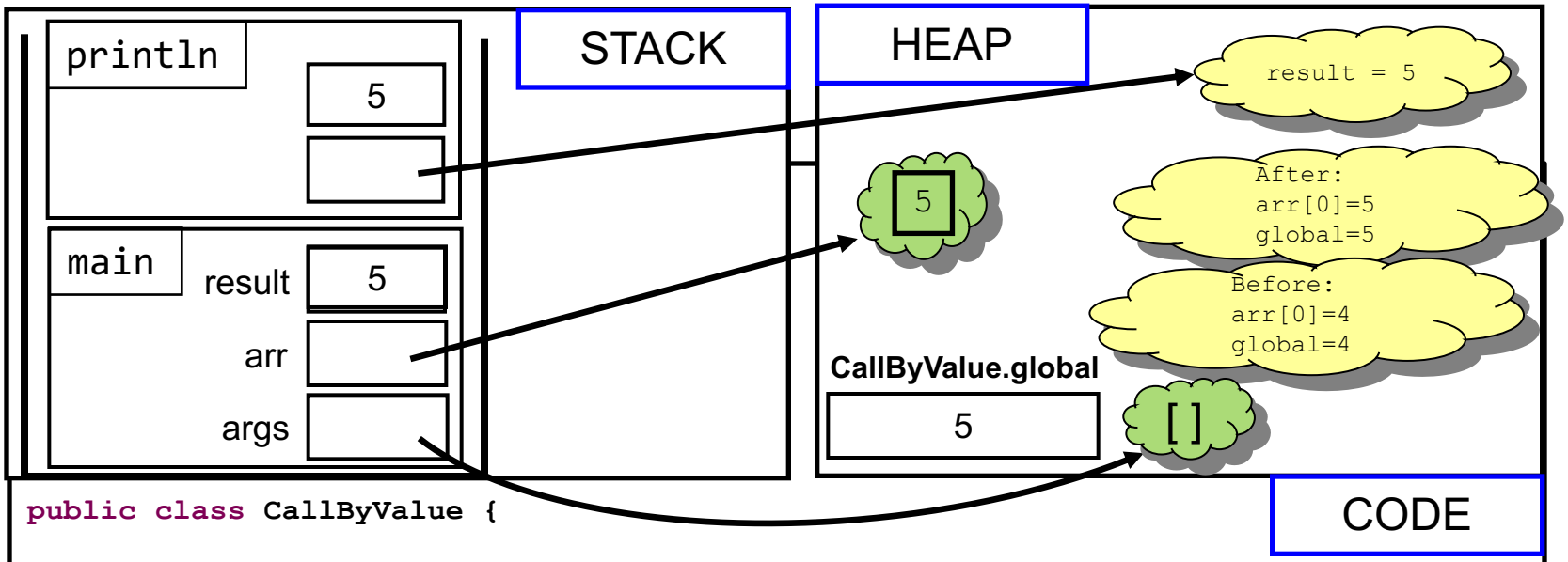



```
public class CallByValue {

     static int global = 4;

     public static int increment(int [] arr){
          int local = 5;
          arr[0]++;
          global++;
          return local;
     }

     public static void main(String[] args) {
          int [] arr = {4};
          System.out.println("Before:\narr[0]=" + arr[0] + "\nglobal=" + global);
          int result = increment(arr);
          System.out.println("After:\narr[0]=" + arr[0] + "\nglobal=" + global);
          System.out.println("result = " + result);
     }
}
```




```
public class CallByValue {

     static int global = 4;

     public static int increment(int [] arr){
          int local = 5;
          arr[0]++;
          global++;
          return local;
     }

     public static void main(String[] args) {
          int [] arr = {4};
          System.out.println("Before:\narr[0]=" + arr[0] + "\nglobal=" + global);
          int result = increment(arr);
          System.out.println("After:\narr[0]=" + arr[0] + "\nglobal=" + global);
          System.out.println("result = " + result);
     }
}
```

Heap, Heap – Hooray!




```
public class CallByValue {

     static int global = 4;

     public static int increment(int [] arr){
          int local = 5;
          arr[0]++;
          global++;
          return local;
     }

     public static void main(String[] args) {
          int [] arr = {4};
          System.out.println("Before: arr[0]=" + arr[0] + "\tglobal=" + global);
          int result = increment(arr);
          System.out.println("After:  arr[0]=" + arr[0] + "\tglobal=" + global);
          System.out.println("result = " + result);
     }
}
```

משתני פלט (Output Parameters)

- איך נכתוב פונקציה שצריכה להחזיר יותר מערך אחד?
  - הפונקציה תחזיר מערך
- ומה אם הפונקציה צריכה להחזיר נתונים מטיפוסים שונים?
  - הפונקציה תקבל כארגומנטים הפניות לעצמים שהוקצו ע"י הקורא לפונקציה (למשל הפניות למערכים), ותמלא אותם בערכים משמעותיים
- ומה קורה אם נרצה שהפונקציה לא תחזיר ערך במקרים מסויימים?
  - החל מ Java 8 – המחלקה Optional עוזרת לדמות את ההתנהגות הרצויה, נראה אותה בהמשך הקורס.

גושי אתחול סטטיים

- ראינו כי אתחול המשתנה הסטטי התרחש מיד לאחר טעינת המחלקה לזיכרון, עוד לפני פונקצית ה main

- ניתן לבצע פעולות נוספות (בדרך כלל אתחולים למניהם) מיד לאחר טעינת המחלקה לזיכרון, פעולות אלו יש לציין בתוך בלוק **`static`**
  - פרטים נוספים:

https://docs.oracle.com/javase/tutorial/java/javaOO/initial.html

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"

Java API

- ניתן למצוא את התיעוד של כל ספריות ה Java באמצעות javadoc באתר של חברת Oracle.

https://docs.oracle.com/javase/8/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 קיימת 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 הגישה לעצמים היא באמצעות טיפוסי הפניה לעצם
  - לא ניתן לגשת לעצם עצמו.
- 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:

```
int i =5 , j = 7;
String s = "Hello", t = "World";
```

i ו- **j** הם מופעים של **int** כשם ש "**hello**" ו- "**world**" הם מופעים של **String**. **s** ו **t** הם הפניות למחרוזות.

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!

שירותי מופע

- בפייתון ניתן לקרוא לשירותי מופע בשתי דרכים:

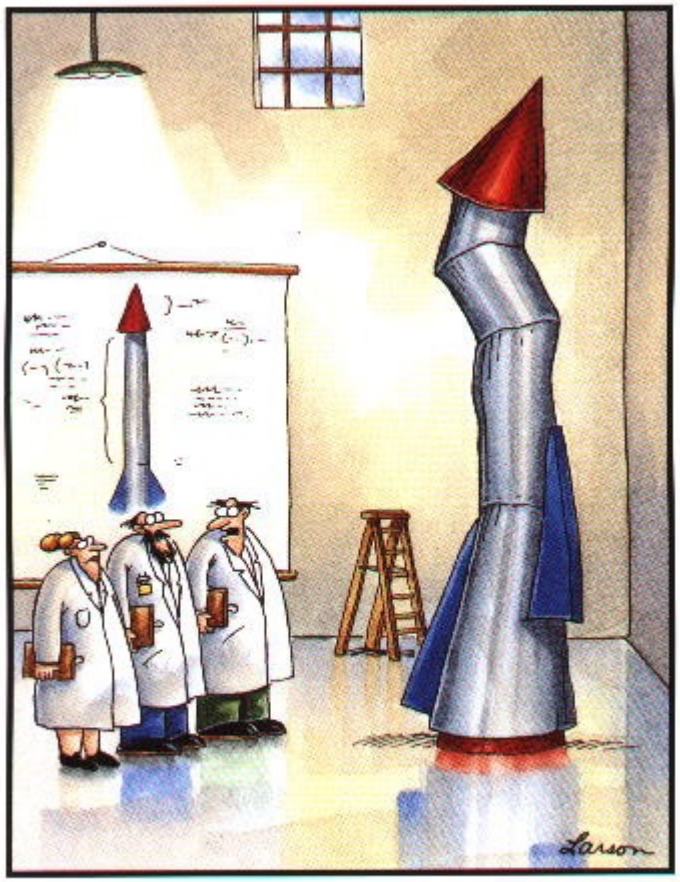
```
myLst = [1,2,3]
myLst.append(4)
list.append(myLst, 5)
```

תזכורת להגדרת המתודה append בתוך list:

```
class list(object):
    def append(self, p_object):
        ...
```

ב Java פונקציה סטטית **לא יכולה לשמש** כפונקציית מופע, ולהיפך!

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 יווצרו בזמן מאוחר יותר – כאשר הלקוח של המחלקה יקרא מפורשות לאופרטור `new`

  - אם יש לנו תבנית לעוגיות, זה לא אומר שיש לנו עוגיות.
  - התבנית מגדירה את הצורה של העוגיות, אבל לא את הטעם שלהן (וניל? שוקולד?)

דוגמא

- נתבונן במחלקה **MyDate** לייצוג תאריכים:

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

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

  - שאלה: כאשר ה JVM טוען לזיכרון את המחלקה איפה בזיכרון נמצאים השדות **day**, **month** ו- **year**?
  - תשובה: הם עוד לא נמצאים! הם יווצרו רק כאשר לקוח ייצר מופע (עצם, אובייקט) 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...
}
```

בהמשך ניראה מימוש אחר של השירותים של MyData.
במימוש זה, לא נצטרך לשלוח את d כפרמטר לשירותים.

נראות פרטית

- מכיוון שהשדות **day**,**month** ו- **year** 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**

- שירותי מופע 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** (syntactic sugar) לשרותי מחלקה, כלומר – 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 main(String[] args (){
        C myC = new C();
        myC.m(args);
}
```

m הוא שירות מופע

```
public class C {

    public static void m(C thisObj, args) {…}

    public static void main(String[] args (){
        C myC = new C();
        C.m(myC,args);
}
```

m הוא שירות סטטי

"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

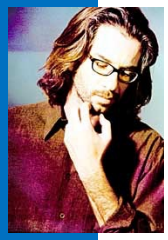
```
public class MyDate {

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

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

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

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

    // more...
}
```

הקוד הזה חוקי !

המשתנה **this** מוכר בתוך שרותי המופע כאילו הועבר ע"י המשתמש.

אולם לא חובה להשתמש בו

```
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 (constructors)

- 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

- המוטיבציה המרכזית להגדרת בנאים היא יצירת עצם שהוא עקבי עם השימוש המיועד שלו (בהמשך נדבר על *משתמר מחלקה* ומצב *מופשט בעל משמעות*)
  - למשל, נרצה שהאובייקט ה **`MyDate`** שאנחנו מייצרים יכיל תאריך חוקי מיד עם יצירתו

```java
public class MyDate {
    private int day;
    private int month;
    private int year;

    public MyDate(int day, int month, int year) {
        this.day = day;
        this.month = month;
        this.year = year;
    }
    // the rest of the code
}
```

הגדרת בנאי ל MyDate

קוד לקוח המשתמש ב- 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());
    }
}
```

בנאים

האם ניתן לאתחל שדה נוסף בתוך הבנאי?

```
public class MyDate {

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

    public MyDate(int day, int month, int year) {
        this.day = day;
        this.month = month;
        this.year = year;
        this.hour = 11;
    }
}
```

בנאים

```
public class MyDate {

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


    public MyDate(int day, int month, int year) {
        this.day = day;
        this.month = month;
        this.year = year;
        this.hour = 11;
    }
}
```

הגדרת שדות

אתחול שדות

לא ניתן לאתחל שדה שלא הוגדר, בדיוק כשם
שלא ניתן לאתחל ערך של משתנה שלא הוגדר!

בנאים

האם ניתן לוותר על השימוש ב **this** בבנאי שהגדרנו?

```
public class MyDate {
    //members here

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

כעת מופיעה בקוד ההשמה הבאה:

```
day=day;
```

בגלל ששם השדה זהה לשם הפרמטר, הורדת השימוש ב **this** מייצרת השמה חסרת משמעות אשר אינה מאתחלת את השדה **day**.

בנאי ברירת מחדל

- במידה ולא הוגדר אף בנאי למחלקה, נוצר בנאי ברירת מחדל (default constructor).
- בנאי ברירת המחדל מתנהג בדיוק כמו הבנאי הבא:

```
public class MyDate {

    public MyDate() {
    }
}
```

- ומאפשר יצירה של אובייקט מטיפוס MyDate באופן הבא:

```
public static void main(String[] args) {
    MyDate d1 = new MyDate();
}
```

בנאים

- לאילו ערכים מאותחלים שדות מחלקה שלא אותחלו בבנאי?

שדות של מחלקות מאותחלים אוטומטית לערכים הדיפולטיים של כל טיפוס (0, null, false), כך שאין חובה לאתחל ערכים אלה בבנאי.

- זכרון שמוקצה על ה Heap מאותחל אוטומטית.

בנאים

- האם הקוד הבא יתקמפל?

```
public class MyDate {

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

    public static void main(String[] args) {
        MyDate d1 = new MyDate();
    }

}
```

בנאי ברירת מחדל נוצר רק כאשר לא הוגדר אף בנאי אחר במחלקה.
אם קיים מימוש של בנאי כלשהו, הבנאי הריק לא נוצר אוטומטית ויש לממש אותו בקוד במידה ונרצה להשתמש בו.

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

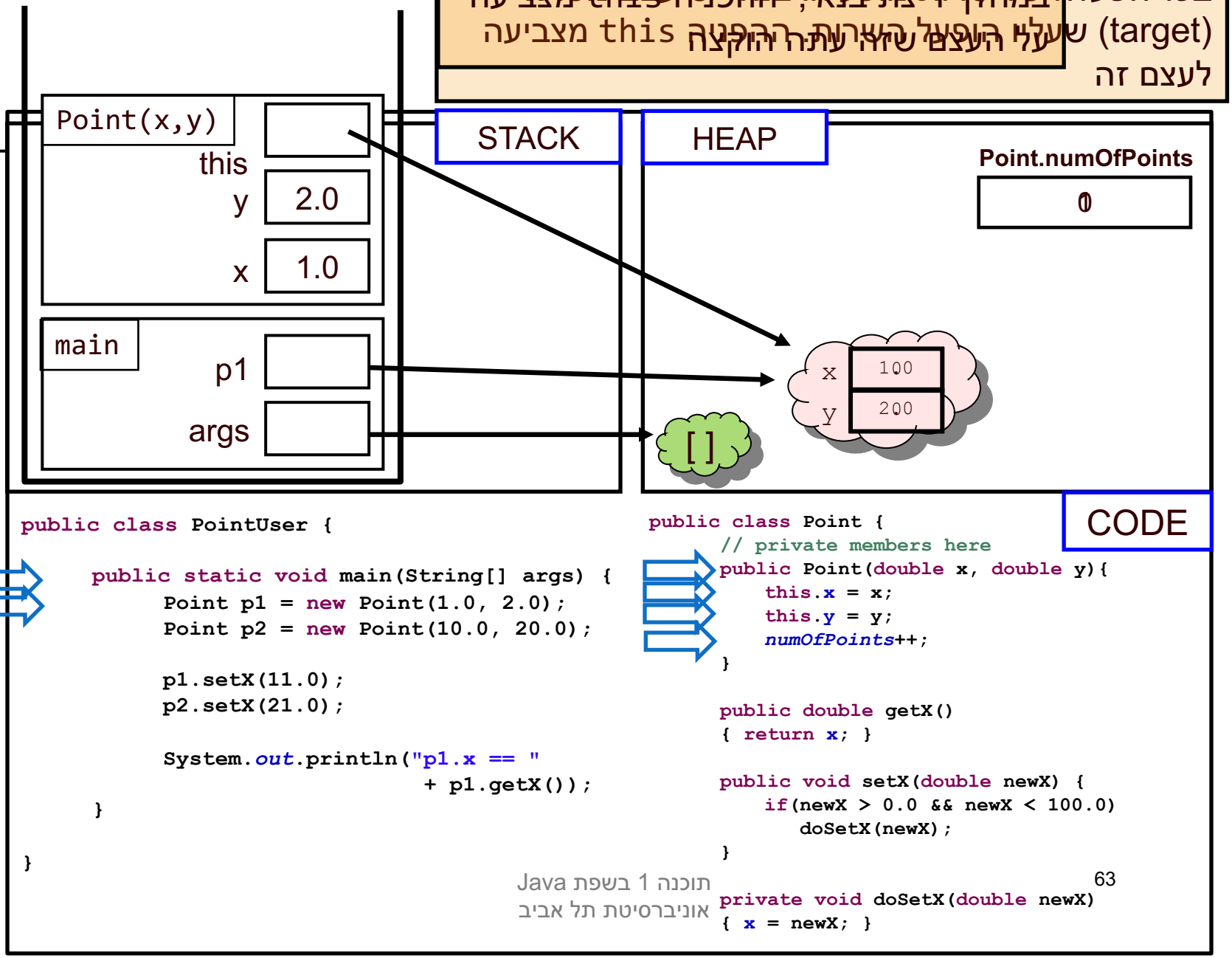
- בהמשך הקורס נציג מימוש מלא ומעניין יותר של המחלקה, אולם כעת לצורך פשטות הדוגמא נסתפק בבנאי, שדה מחלקה, שני שדות מופע ושלושה שרותי מופע

```
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;
    }

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

    private void doSetX(double newX) {
       x = newX;
    }

    // More methods...
}
```

PointUser

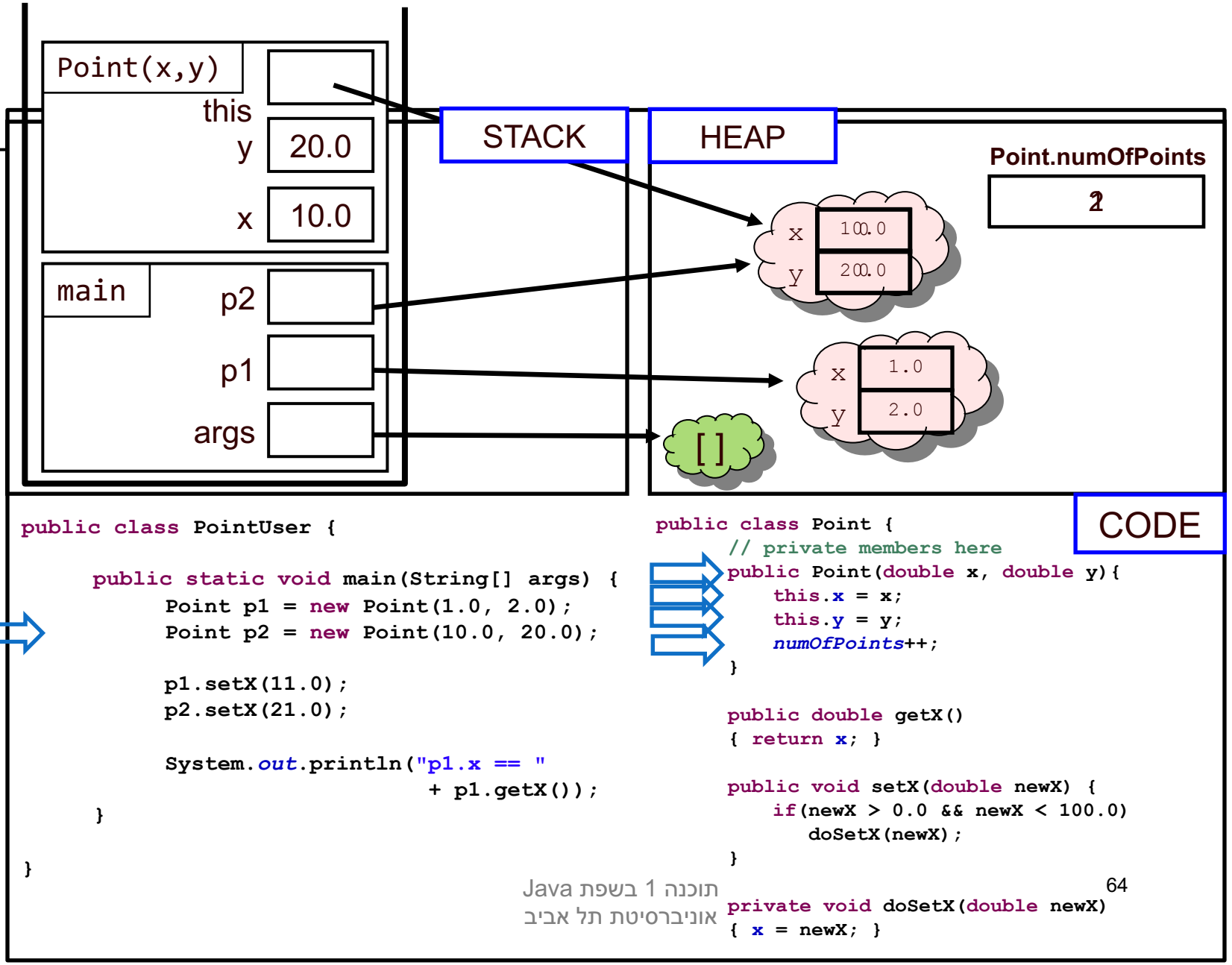
```
public class PointUser {

    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());
    }

}
```

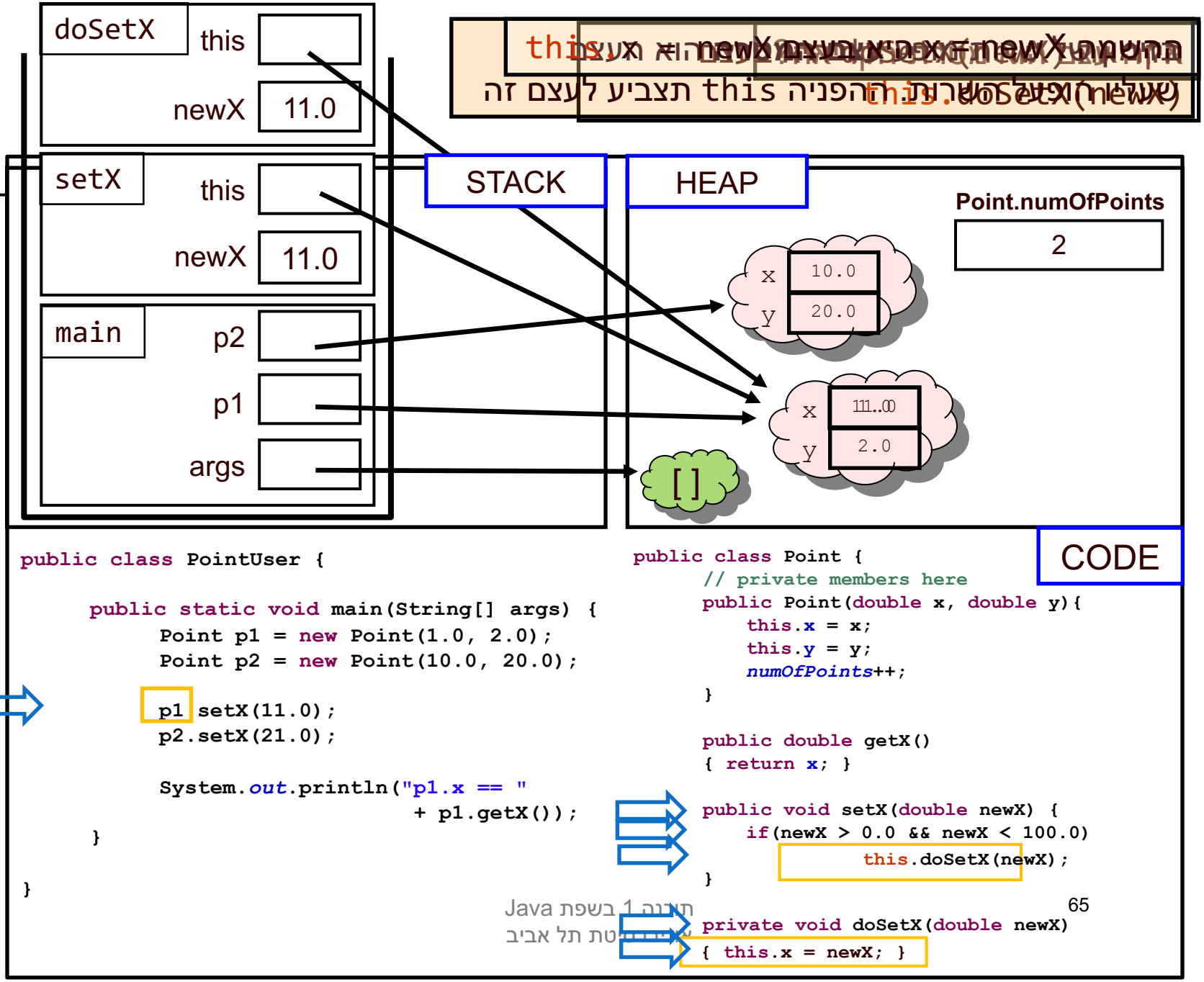
בכל הפעלה של שיטת מופע או בנאי, ההפניה this מצביעה על העצם שזה עתה הוקצה
במהלך ריצת בנאי, ההפניה this מצביעה על העצם שזה עתה הוקצה
(target) שעליו הופעל השירות, ההפניה this מצביעה לעצם זה

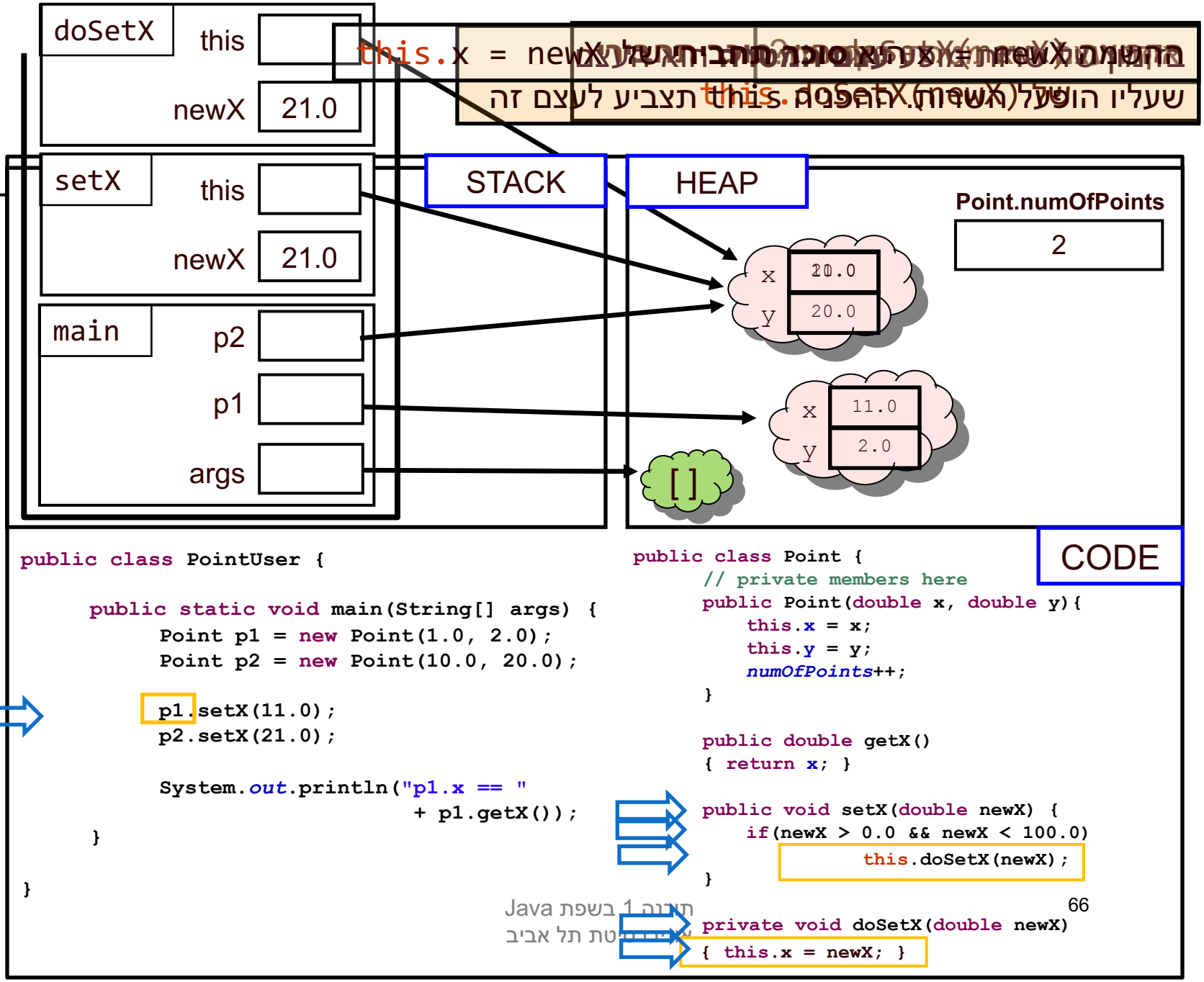



```
public class PointUser {

    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());
    }

}
```

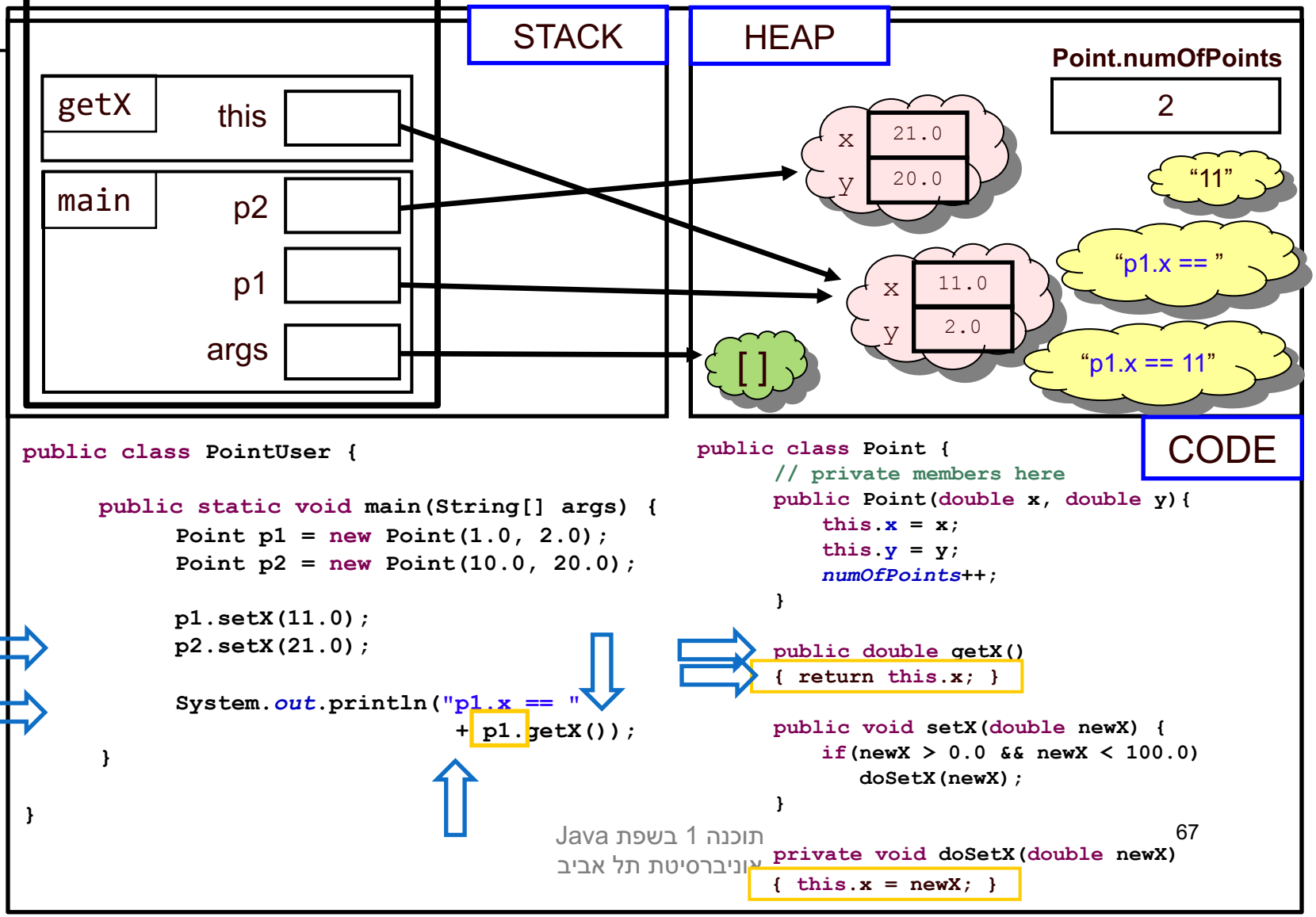
```
public class Point {
    // private members here
    public Point(double x, double y){
        this.x = x;
        this.y = y;
        numOfPoints++;
    }

    public double getX()
    { return x; }

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

    private void doSetX(double newX)
    { x = newX; }
```


Point(x,y)
this
y
20.0
x
10.0
main
p2
p1
args
STACK
HEAP
Point.numOfPoints
2
x
100.0
y
200.0
x
1.0
y
2.0
[ ]
CODE
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 {
// private members here
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);
}
private void doSetX(double newX)
{ x = newX; }


doSetX
this
newX
11.0
setX
this
newX
11.0
main
p2
p1
args
STACK
HEAP
Point.numOfPoints
2
x
10.0
y
20.0
x
111..00
y
2.0
[]
CODE
this.x = newX; // הפניה this תצביע לעצם זה
this.doSetX(newX);
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 {
// private members here
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);
}
private void doSetX(double newX)
{ this.x = newX; }


doSetX
this
newX
21.0
this.x = newX;
setX
this
newX
21.0
main
p2
p1
args
STACK
HEAP
Point.numOfPoints
2
x
y
20.0
x
y
11.0
2.0
[]
CODE
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 {
// private members here
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);
}
private void doSetX(double newX)
{ this.x = newX; }

"return this.x" מה יודפס? מה יודפס "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 {
      // private members here
      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);
      }

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

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