תוכנה 1 בשפת Java

שיעור מספר 5: עצמים

שחר מעוז

בית הספר למדעי המחשב

אוניברסיטת תל אביב

על סדר היום

- הגדרת טיפוסים (מחלקות) חדשים
- מודל הזיכרון של זימון שרותי מופע
- כתיבת מחלקות חדשות לפי מפרט
- מצב מופשט של עצם

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**
  - 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 בזיכרון 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?
- 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

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

```java
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...
}
```

```java
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

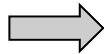
- מכיוון שהשדות **day ,month** ו- **year** הוגדרו בנראות פרטית (**private**) לא ניתן להשתמש בהם מחוץ למחלקה (שגיאת קומפילציה)

```
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
- 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

```java
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...
}
```

הקוד הזה חוקי !

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

אולם לא חובה להשתמש בו

```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 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)

```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

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

      // More methods...
}
```

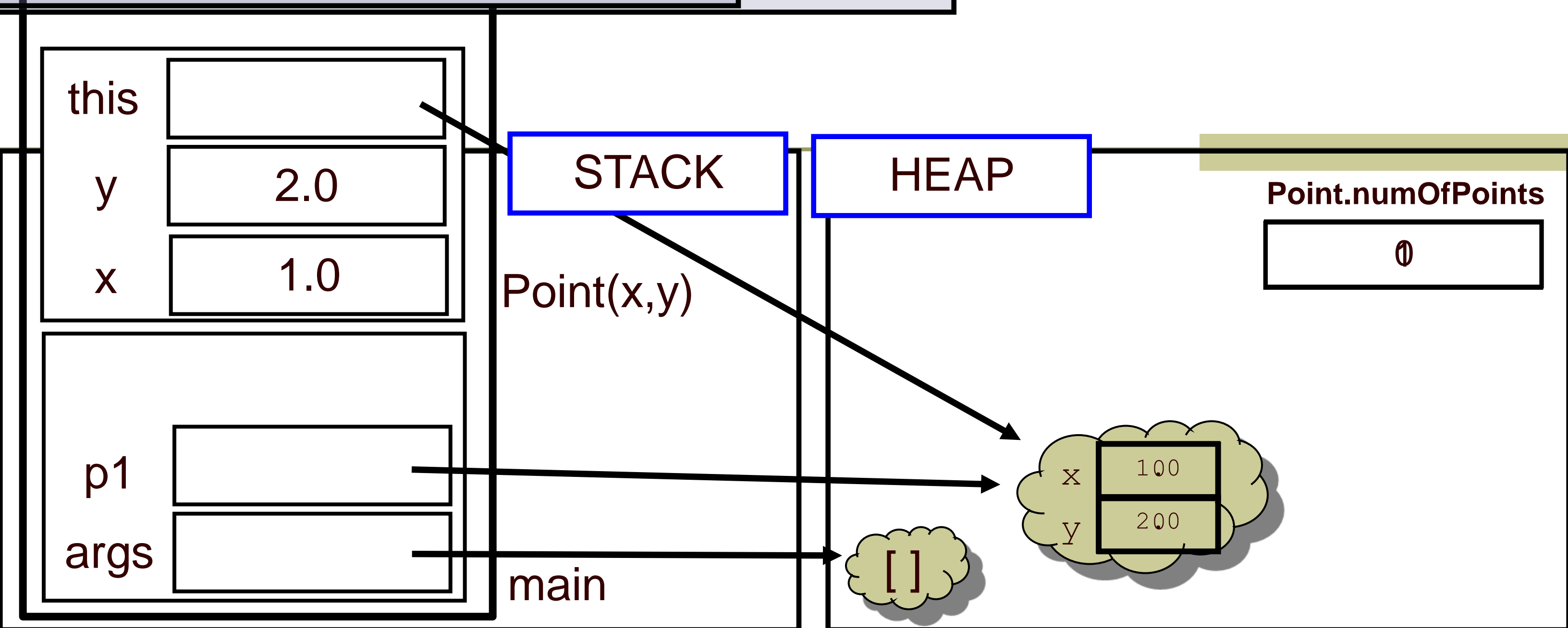
PointUser

```
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

בכל הפעלה של שרות מופע יש עצם מטרה (target)
שעליו הופעל השרות. ההפניה this מצביעה לעצם זה

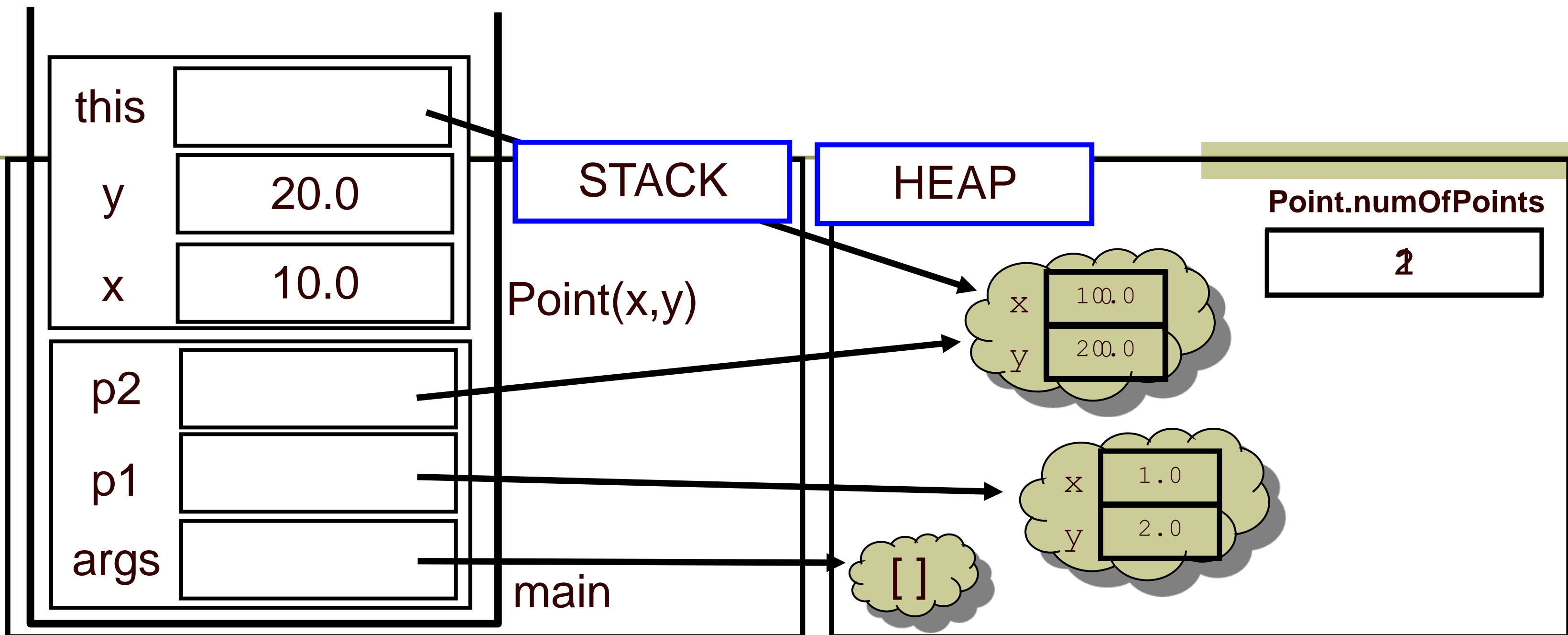
במהלך ריצת בנאי, ההפניה this מצביעה
על העצם שזה עתה הוקצה



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());
    }

}
```

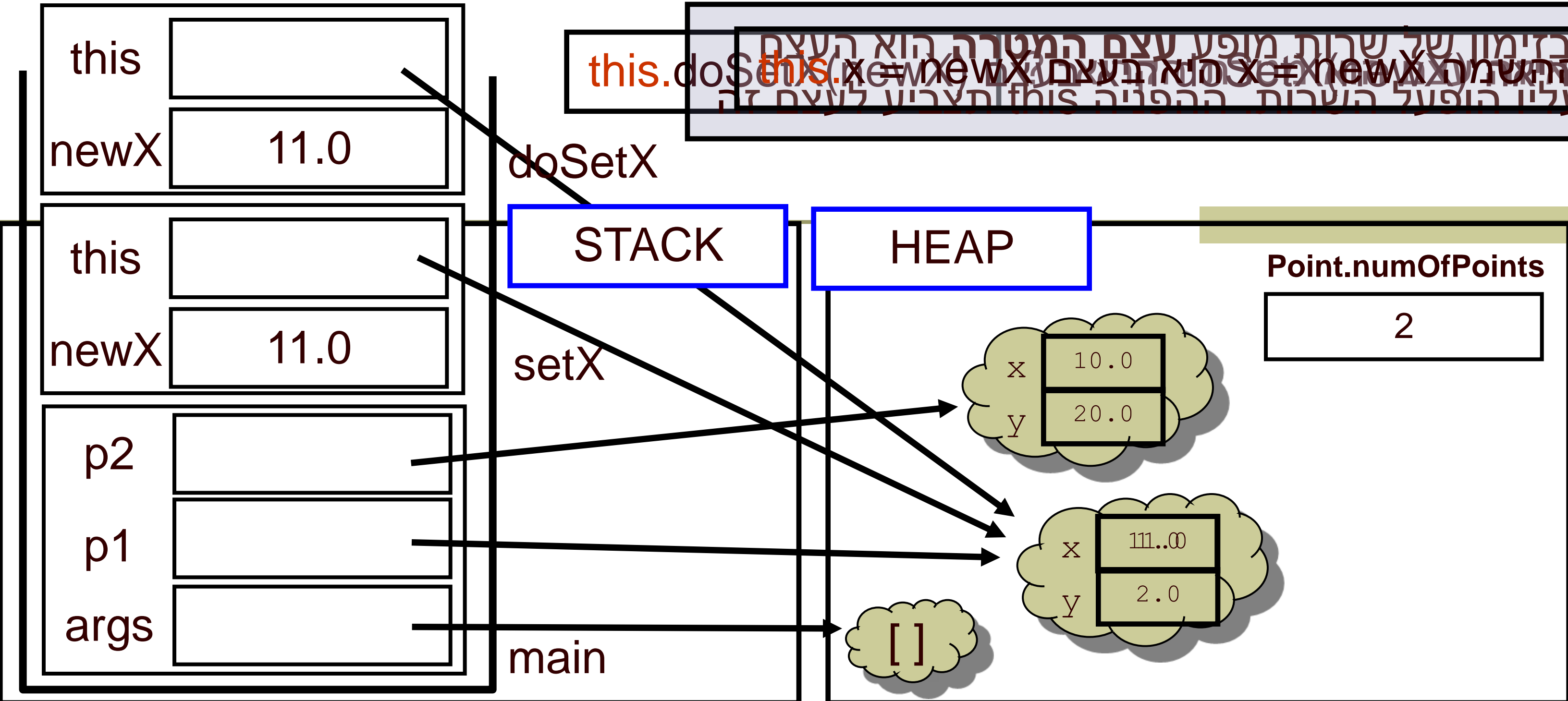
```
public class Point {

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; }
```




```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




```
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

this.x = newX

ההשמה x = newX היא סוכר תחבירי של

this		doSetX
newX	21.0	
this		setX
newX	21.0	
p2		main
p1		
args		

STACK

HEAP

Point.numOfPoints

2

x 10.0 / 21.0

y 20.0

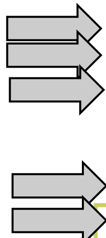
x 11.0

y 2.0

[ ]

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; }
```

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

STACK

getX

this

p2

p1

args

main

HEAP

Point.numOfPoints

2

x 21.0
y 20.0

x 11.0
y 2.0

[ ]

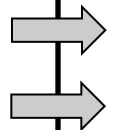
"11"

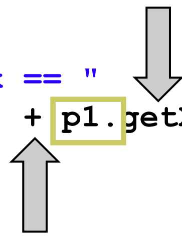
"p1.x == "

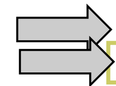
"p1.x == 11"

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 this.x; }

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

    public 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

עוד על עצמים ותמונת הזיכרון

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

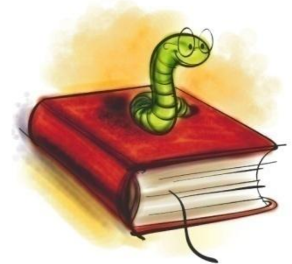
title	"The Red and the Black"
date	1830
page_count	341

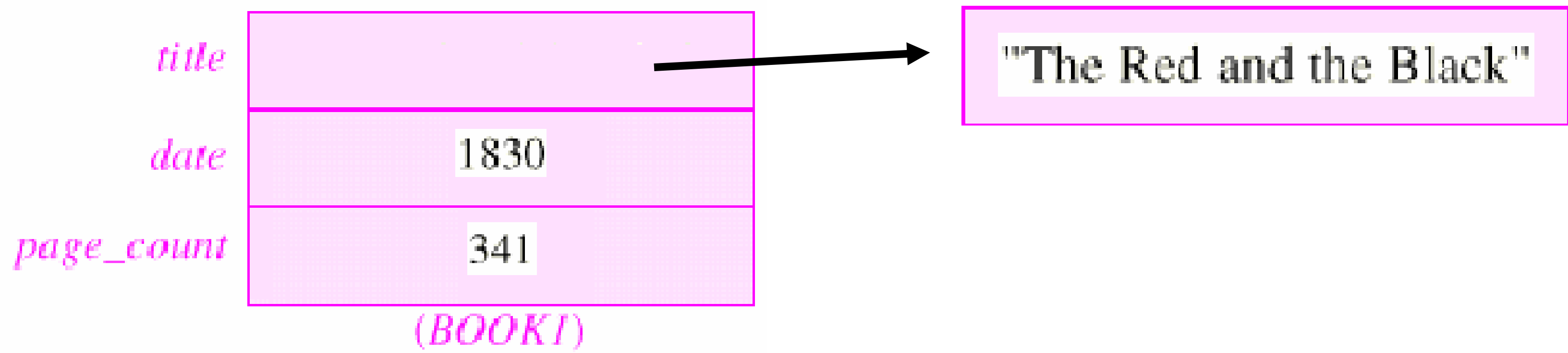
(BOOK1)

<u>התרשים פשטני</u> – *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;
}
```

name	"Stendhal"
real_name	"Henri Beyle"
birth_year	1783
death_year	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)

title	"The Red and the Black"
date	1830
page count	341

name	"Stendhal"
real_name	"Henri Beyle"
birth_year	1783
death_year	1842

(WRITER)

(BOOK2)

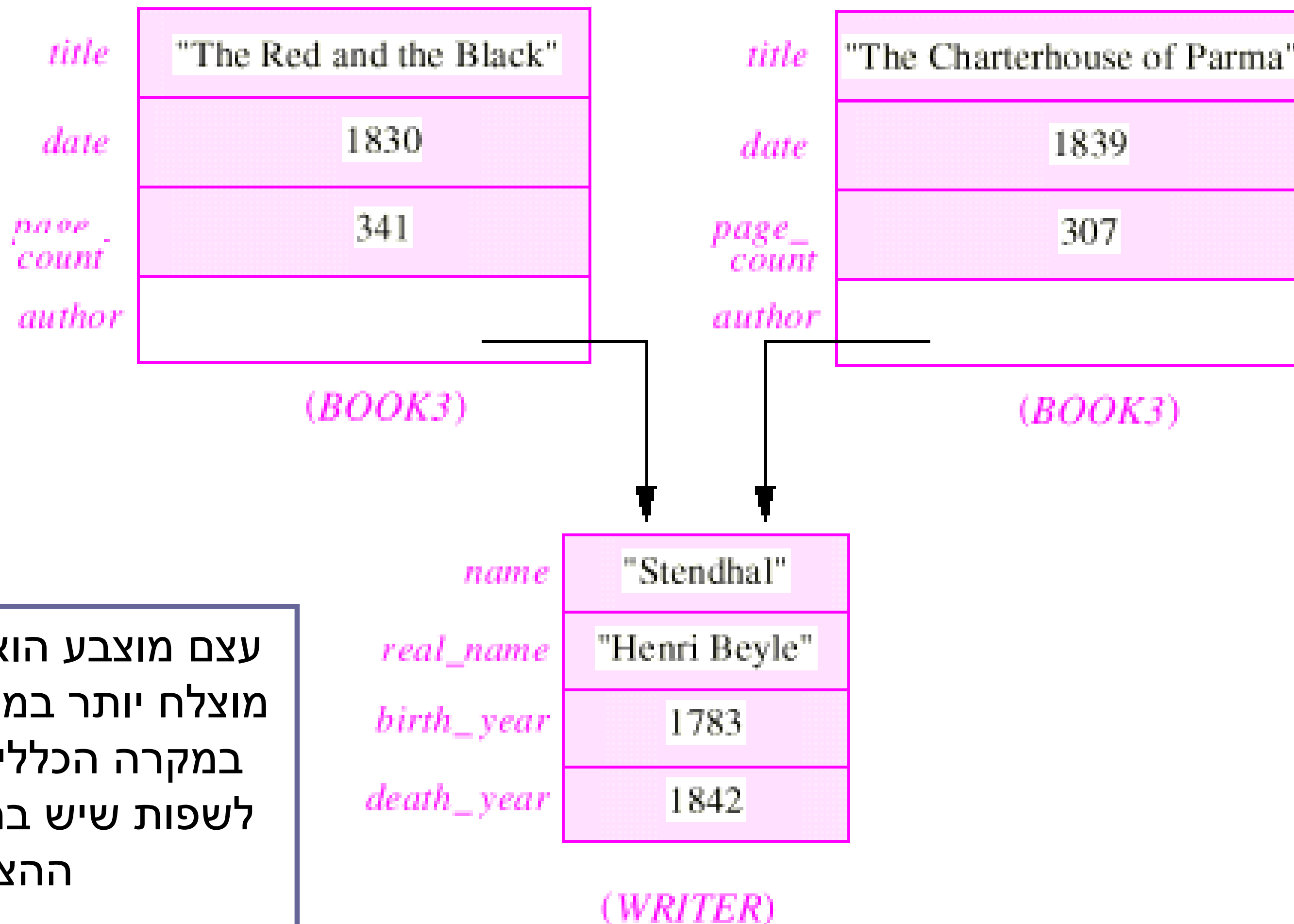
title	"Life of Rossini"
date	1823
page count	307

name	"Stendhal"
real_name	"Henri Beyle"
birth_year	1783
death_year	1842

(WRITER)

(BOOK2)

עצם מוצבע



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

Coding using a specification
כתיבת מחלקות חדשות לפי מפרט

נכונות של מחלקות

- קיימות כמה גישות לפיתוח של קוד בד בבד עם המפרט שלו (specification) – בקורס נציג שילוב של שתיים מהן
- פרט לציון החוזה של כל שרות (פונקציה) ושל המחלקה כולה בעזרת טענות בולאניות (**D**esign **b**y **C**ontract- DbC) נגדיר לטיפוס הנתונים **מצב מופשט ופונקצית הפשטה**
- נציין **גישה אחרת** הטוענת כי תחילה יש להגדיר ADT (**A**bstract **D**ata **T**ype) – טיפוס נתונים מופשט, וממנו לגזור טענות DbC (**D**esign **b**y **C**ontract)

הגדרת מחסנית של שלמים

- נרצה להגדיר מבנה נתונים המייצג מחסנית של מספרים שלמים עם הפעולות: push, pop, top, isEmpty
- מחסנית היא מבנה נתונים העובד בשיטת LIFO
  - כפי שעובד מקרר, ערמת תקליטורים או מחסנית נשק

```
StackOfInts s1 = new StackOfInts();
System.out.println("isEmpty() == " + s1.isEmpty());   // true
s1.push(1);
System.out.println("s1.top() == " + s1.top());       // 1
s1.push(2);
System.out.println("s1.top() == " + s1.top());       // 2
s1.pop();
System.out.println("s1.top() == " + s1.top());       // 1
System.out.println("isEmpty() == " + s1.isEmpty());   // false
```

- נציג חוזה לטיפוס המחסנית

מה יקרה אם כעת ננסה לבצע s1.top() ?

```java
public class StackOfInts {

    /**
     * @post isEmpty() , "The constructor creates an empty stack" */
    public StackOfInts() { ... }

    /** returns top element
     *  @pre !isEmpty() , "can't top an empty stack" */
    public int top() { ... }

    /** returns true when the stack is empty */
    public boolean isEmpty() { ... }

    /** removes top element
     *  @pre !isEmpty() , "can't pop an empty stack" */
    public void pop() { ... }

    /** adds x to the stack as top element
     *  @post top() == x , "x becomes top element"
     *  @post !isEmpty() , "Stack can't be empty" */
    public void push(int x) { ... }

}
```

בעיה: החוזה "שטוח" ואינו מבטא את מהות הפעולות

הצעה לפתרון: נוסיף עוד שאילתה count() שתחזיר את מספר האברים שבמחסנית

```
/** @inv count() >= 0 */
public class StackOfInts {

    /**
     * @post isEmpty() , "The constructor creates an empty stack" */
    public StackOfInts() { ... }

    /** returns top element
     * @pre !isEmpty() , "can't top an empty stack" */
    public int top() { ... }

    /** returns top element
     *  @post $ret == (count() == 0) */
    public boolean isEmpty() { ... }

    /** removes top element
     * @pre !isEmpty() , "can't pop an empty stack"
     *  @post count() == $prev(count()) - 1 */
    public void pop() { ... }

    /** adds x to the stack as top element
     * @post top() == x , "x becomes top element"
     * @post !isEmpty() , "Stack can't be empty"
     *  @post count() == $prev(count()) + 1 */
    public void push(int x) { ... }

    /** returns the number of elements in the stack*/
    public int count() { ... }
}
```

הפתרון בעייתי:

1. המתודה count() אינה חלק מהקונספט של מחסנית
2. גם בעזרתה לא ניתן לתאר את המהות שבפעולות
3. עדיף היה לשמור את ההשפעה על count לחוזה המימוש של המחלקה

ננסה לחשוב על תאור מופשט (פשטני, פשוט) של טיפוס הנתונים כדי שנוכל על פיו לתאר את משמעות 5 הפעולות

ניסוח המצב המופשט (abstract state)

- נגסח את הטיפוס שאותו רוצים להגדיר בצורה מדוייקת, פשטנית, אולי מתמטית אבל לא בהכרח (לפעמים תרשים יכול להיות פשוט יותר ומדויק לא פחות)
- כל התכונות ינוסחו במונחי התאור המופשט. **החוזה** של שרותי המחלקה יבוטא בעזרת התמרות (transformations) או מאפיינים של **המצב המופשט**
- בשקף הבא נציג תאור של המצב המופשט בעזרת שימוש בתגית @abst – זהו **תחביר משלים** לניסוח החוזה שראינו בשקף הקודם
- מסובך? דווקא פשוט. פשטני.

```
/**  @abst (i_1, i_2, ... , i_n) or () for the empty stack */
public class StackOfInts {

    /** @abst AF(this) == () */
    public StackOfInts(){ ... }

    /**  @abst $ret == i_1  */
    public int top() { ... }

    /** @abst $ret == (AF(this) == ()) */
    public boolean isEmpty() { ... }

    /** @abst AF(this) == (i_2, i_3, ... , i_n) */
    public void pop() { ... }

    /** @abst AF(this) == (x, i_1, ... , i_n) */
    public void push(int x) { ... }

    /** @abst $ret == n */
    public int count() { ... }
}
```

מצב וערך מוחזר במונחים מופשטים

- עבור **פקודות ובנאים**, התיאור מציין מהו המצב המופשט החדש, לאחר ביצוע הפקודה

```
@abst AF(this) == (i2, i3, ... , in)
```

- עבור **שאילתות**, התיאור מציין מהו הערך שיוחזר

```
@abst $ret == i1
```

  - שאילתא אינה משנה את המצב המופשט

- הכל ביחס **למצב המופשט** שהיה לפני השרות, כפי שמופיע בראש המחלקה

```
@abst (i1, i2, ... , in)
```

מצב מופשט ועצם מוחשי

- בהינתן מפרט (חוזה + מצב מופשט) ייתכנו כמה מימושים שונים שיענו על הדרישות
- בחירת המימוש מביאה בחשבון הנחות על אופן השימוש במחלקה
- בחירת המימוש מונעת משיקולי יעילות, צריכת זיכרון ועוד
- לאחר בחירת מימוש נציג **פונקצית הפשטה** שתמפה כל טיפוס קונקרטי (עצם בתוכנית) למצב מופשט בהתאם לייצוג שבחרנו
- כדי להוכיח את **נכונות המימוש** נוכיח כי המימושים של כל השרותים עקביים (consistent) עם המצב המופשט

מימוש אפשרי ל StackOfInts

```
public class StackOfInts {

    public static int DEFAULT_STACK_CAPACITY = 10;

    private int [] rep;
    private int count;

    public StackOfInts(){
        count = -1;
        rep = new int[DEFAULT_STACK_CAPACITY];
    }
```

* השמטנו את החוזה והמצב המופשט בגלל מגבלות השקף...

מימוש אפשרי ל StackOfInts (המשך)

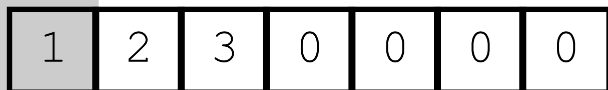
```
public int top(){
   return rep[count];
}

public boolean isEmpty(){
   return count == -1;
}

public void pop(){
   count--;
}

public int count(){
   return count + 1;
}
```

מימוש אפשרי ל StackOfInts (המשך 2)

```
public void push(int x){
    if (count == rep.length - 1)
        enlargeRep();
    count++;
    rep[count] = x;
}

/** allocate storage space in rep */
private void enlargeRep(){
    int [] biggerArr = new int[rep.length * 2];
    System.arraycopy(rep, 0, biggerArr, 0, rep.length);
    rep = biggerArr;
}

}
```

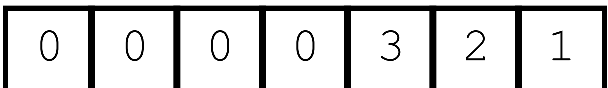
מימוש חלופי ל StackOfInts

- במימוש שראינו בחרנו לייצג את הנתונים בעזרת מערך
  - מילאנו את האברים מהמקום ה-0 ואילך ורוקנו את האיברים מהמקום האחרון קדימה ע"י הקטנת count



count

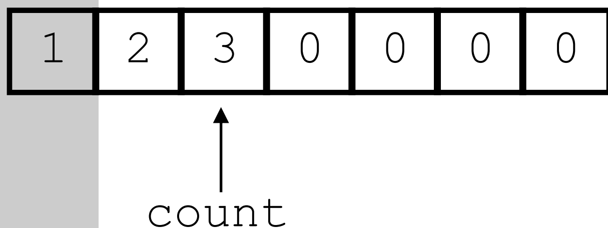
- יכולנו לנקוט גישה אחרת:
  - למלא את האברים מהמקום האחרון לראשון ולרוקן אותם ע"י הגדלת count



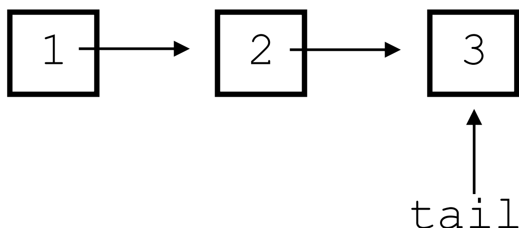
count

מימוש חלופי ל StackOfInts

- כותב המחלקה **StackOfInts** מטפל בהגדלת המערך כאשר הוא מתמלא
  - בעזרת הפונקציה הפרטית **enlargeRep** המקצה מקום חדש כפול ומעתיקה את המערך לשם



- יכולנו לנקוט גישות אחרות:
  - להשתמש ברשימה מקושרת של תאים
  - להשתמש במבני נתונים הגדלים דינאמית



שימוש ב- private להפחתת התלות לקוח-ספק

- כאשר אין גישה לשדות פנימיים של המחלקה יכול הספק להחליף בהמשך את מימוש המחלקה בלי לפגוע בלקוחותיו
  - למשל אם נרצה בעתיד להחליף את המערך ברשימה מקושרת או להחליף את סדר הכנסת האברים
- שדה מופע שנחשף ללקוחות (שאינו private) יהיה חייב להיות נגיש להם ובעל ערך עדכני בכל גירסה עתידית של המחלקה כדי לשמור על תאימות לאחור של המחלקה
- **לכן תמיד נסתיר את הייצוג הפנימי מלקוחותינו**

javadoc ונראות

- כלי התיעוד javadoc תומך בדרגות ניראות שונות
- כברירת מחדל, במסמך התיעוד הנוצר אין אזכור של מרכיבי המחלקה הפרטיים (אפילו לא שמם!)
- ניתן להגדיר את דרגת הנראות בעת יצירת התיעוד, וכך להפיק מסמכי תיעוד שונים למפתחי המחלקה וללקוחות המחלקה (אולי מפתחים בעצמם)

פונקצית ההפשטה (abstraction function)

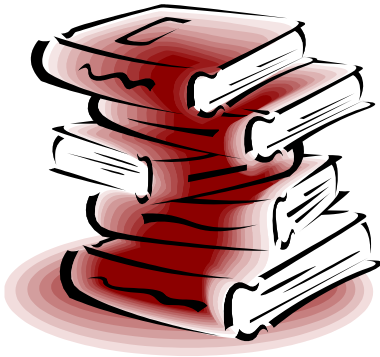
- ראינו כי קיימות דרכים רבות לייצג (לממש) מחלקה
- בחירת הייצוג נקרא **שלב העיצוב** או **שלב התיכון** של המחלקה (design phase)
- לאחר שבחרנו ייצוג למחלקה אנו צריכים להיות עקביים במימוש כדי שהמימוש יהיה תואם למפרט
- לצורך כך עלינו לנסח **פונקצית הפשטה, AF**, הממפה מימוש קונקרטי (ייצוג בזיכרון התוכנית, **this**) למצב מופשט **AF(this)**
- פונקצית ההפשטה היא במובנים רבים **התהליך ההופכי** לתהליך העיצוב

פונקצית ההפשטה ל StackOfInts

$$AF(this) \equiv (x_1, \ldots, x_n)$$

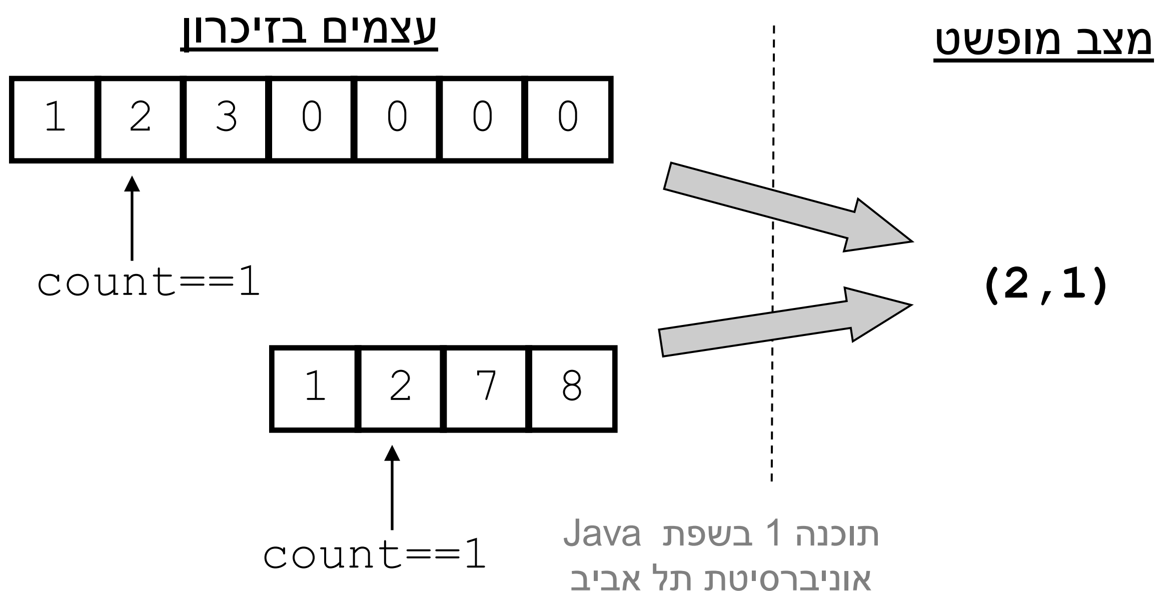
$$s.t.: \forall i = 1..n : x_i = rep[count - i + 1],$$

$$n = count + 1$$



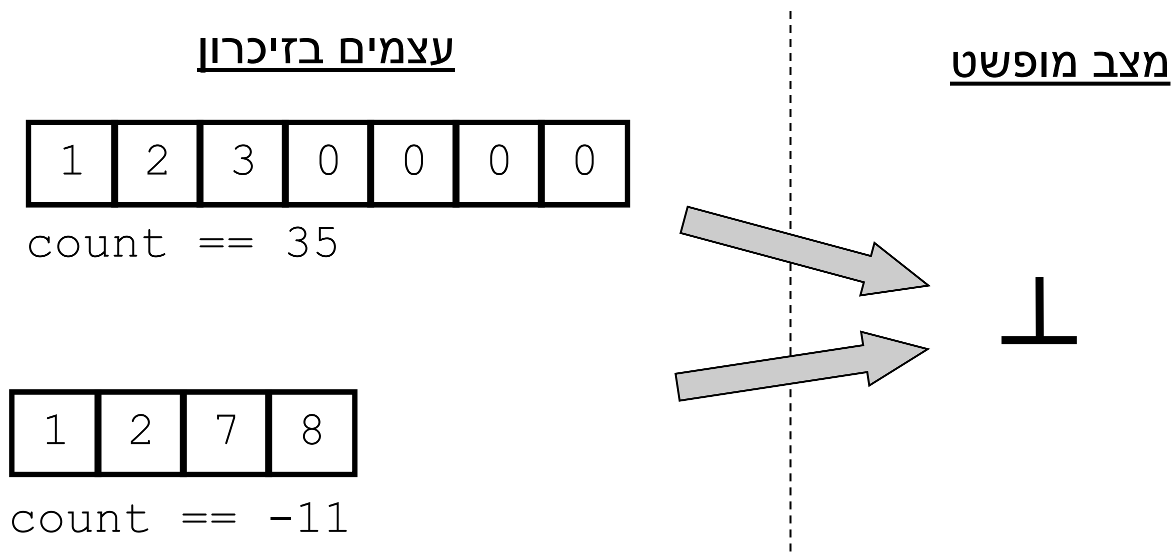
פונקצית ההפשטה אינה חד-חד ערכית

- פונקצית ההפשטה **הינה חד-ערכית** אולם בדרך כלל **אינה חד-חד ערכית**:
  - בהינתן מימוש של מחלקה יתכנו עצמים במצבים מוחשים שונים (תמונת זיכרון שונה, concrete state) אשר ימופו לאותו מצב מופשט



פונקצית ההפשטה אינה מלאה (partial)

- קיימים מצבים מוחשים שאינם חוקיים, כלומר לא ניתן למפות אותם לאף מצב מופשט תקין



מִשְׁתָּמֶר הייצוג

- במהלך חייו של עצם, מכיוון שבכל רגע נתון הוא אמור לייצג מצב מופשט כלשהו, קיימים אילוצים על הערכים של שדותיו

- אילוצים אלו נקראים משתמר הייצוג (representation invariant) והם צריכים להתקיים "תמיד". כלומר:
  - בסיום הבנאי
  - בכניסה לכל שירות ציבורי וביציאה מכל שירות ציבורי

הוכחת נכונות של מחלקה

- **שלב א':** נוכיח כי כאשר נוצר עצם חדש, הוא מקיים את משתמר הייצוג
- **שלב ב':** עבור כל שירות במחלקה נוכיח: אם מתקיים בכניסה לשירות תנאי הקדם וגם המשתמר מתקיים, אזי ביציאה מהשירות מתקיים תנאי האחר וגם המשתמר מתקיים
- **שלב ג':** נוכיח כי פרט לשירותים של המחלקה, אין בתוכנית קוד שעשוי להפר את המשתמר אם הוא כבר מתקיים
  - בדוגמא שלנו – אף אחד לא יכול 'להתעסק' עם `rep` ו- `count` מחוץ למחלקה

משתמר הייצוג של StackOfInts

```
/** @imp_inv count < rep.length
 *  @imp_inv count >= -1
 *  @imp_inv isEmpty() || top() == rep[count]
 *  @imp_inv isEmpty() == (count==-1)
 */
public class StackOfInts {
```

- חלק מהטענות יכולות להופיע גם בתור תנאי בתר מימושי (**imp_post**) של השאילתות המתאימות.
  למשל הטענה `@imp_inv isEmpty() || top() == rep[count]` שקולה ל:

```
/** @imp_post $ret == rep[count] */
public int top() { ... }
```

הוכחת נכונות של מחלקה

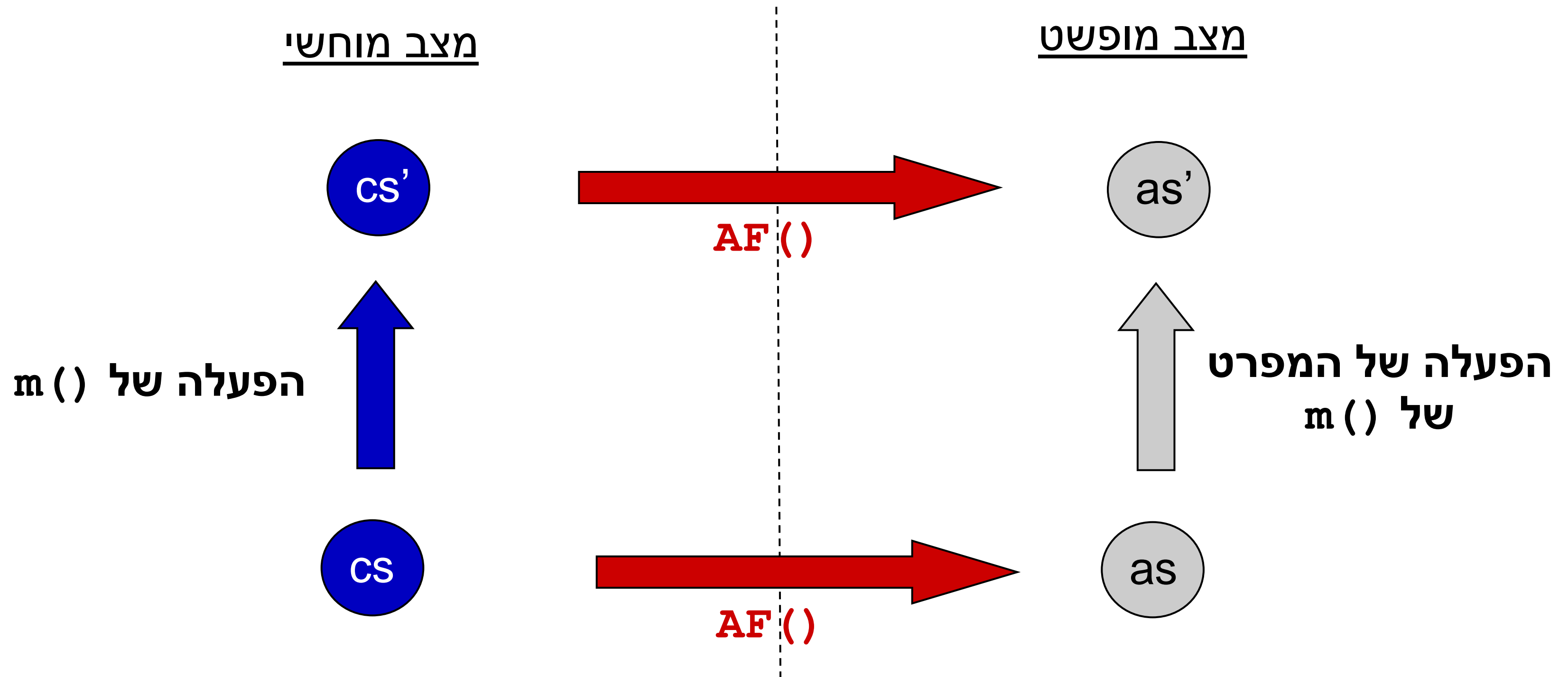
- אולם לא מספיק להראות כי השרותים והבנאים משרים על העצמים ערכים **חוקיים**, צריך גם להראות כי כל השרותים **עושים מה שהם צריכים** לעשות
  - כלומר מימוש השרותים עקבי עם ההפשטה שנבחרה
  - ומתקיים החוזה של כל השרותים והבנאים

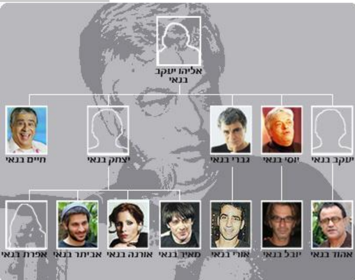
- נכונות של שרות (פקודה) `m()`:
  - בהינתן מצב מופשט **as** ופקודה `m()` המתמירה אותו למצב מופשט **as’** צריך להתקיים כי עבור עצם עם מצב מוחשי **cs** (הממופה ל- **as**) השרות `m()` מעביר אותו למצב **cs’** הממופה ל **as’**

```
AF(cs.m()) == AF(cs).m()
```

נכונות המימוש

- כלומר שני המסלולים בתרשים שקולים:





העמסת בנאים (constructor overloading)

- כדי שעצם שזה עתה נוצר יקיים את המשתמר יש לממש לו בנאי מתאים
- ניתן להעמיס בנאים בדומה להעמסת פונקציות
- דוגמא: כדי לחסוך הכפלות מערכים עתידיות נרצה להקצות מראש מערך בגודל המצופה

```
public class StackOfInts {

    public static int DEFAULT_STACK_CAPACITY = 10;

    public StackOfInts(){
        count = -1;
        rep = new int[DEFAULT_STACK_CAPACITY];
    }

    public StackOfInts(int expectedCapacity){
        count = -1;
        rep = new int[expectedCapacity];
    }
```

- חסרונות המימוש: שכפול קוד! אם בעתיד נחליף את הייצוג או המימוש שכפול הקוד עשוי לאבד את עיקביותו

העמסת בנאים נכונה

- נאכוף את העקביות ע"י קריאה הדדית בין הבנאים
- בהעמסת בנאים אם אחת מהגרסאות המועמסות תרצה לקרוא לגרסה אחרת עליה להשתמש במבנה **this(*args*)**

```
public class StackOfInts {

    public static int DEFAULT_STACK_CAPACITY = 10;

    public StackOfInts(){
       this(DEFAULT_STACK_CAPACITY);
    }

    public StackOfInts(int expectedCapacity){
       count = -1;
       rep = new int[expectedCapacity];
    }
```

- בשפת Java השימוש ב **this(*args*)** אם קיים, חייב להופיע בשורה הראשונה של הבנאי