תוכנה 1 בשפת Java
שיעור מספר 4: "זרוק לו עצם"

סיון טולדו
אוהד ברזילי

בית הספר למדעי המחשב
אוניברסיטת תל אביב

על סדר היום

- חוזים, נכונות והסתרת מידע
- מחלקות כטיפוסי נתונים
- שימוש במחלקות קיימות
- כתיבת מחלקות חדשות
- הוכחת נכונות של מחלקות

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 המשתמש אינו **`true`**

הוכחת נכונות של טענה

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

```
/** @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

```java
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

```java
/** @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
- 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 המשתמשת 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 ראינו בתרגול וכן 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)
- שרותי מופע הם פונקציות המופנות **לעצם מסוים** ומבקשות ממנו בקשה, או שואלות אותו שאלה

שימוש במחלקות קיימות

- לטיפוס מחלקה תכונות בסיסיות, אשר סיפק כותב המחלקה, ואולם ניתן לבצע עם העצמים פעולות מורכבות יותר ע"י שימוש באותן תכונות
- את התכונות הבסיסיות יכול הספק לציין למשל בקובץ תיעוד
- תיעוד נכון יתאר **מה** השרותים הללו עושים ולא **איך** הם ממומשים
- התיעוד יפרט את חתימת השרותים ואת החוזה שלהם

- נתבונן במחלקה Turtle המייצגת **צב לוגו** המתקדם על משטח ציור
  - כאשר זנבו למטה הוא מצייר קו במסלול ההתקדמות
  - כאשר זנבו למעלה הוא מתקדם ללא ציור

- כותבת המחלקה לא סיפקה את הקוד שלה אלא רק עמוד תיעוד המתאר את הצב (המחלקה ארוזה ב JAR של קובצי class)

Turtle API

Turtle - Microsoft Internet Explorer

File Edit View Favorites Tools Help

Back Search Favorites

Address E:\Ohady\courses\advanced java\wernerer05\Exercises\Ex1\ex1\API\Turtle.html

Class Turtle

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

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

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

Constructor Summary

Turtle()
constructs a new turtle

Method Summary

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

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

שרותים – נפריד בין 2 סוגים
שונים:

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

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

Turtle API

Method Summary

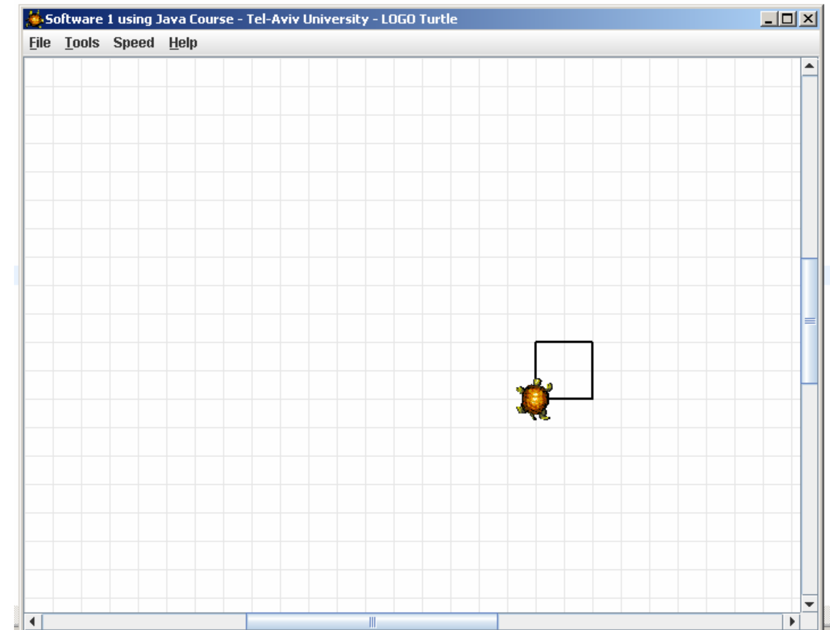
double	getAngle() returns the direction which the turtle is facing
static int	getDelay() return the delay the turtle
double	getX() returns the x coordinate of the turtle's location
double	getY() returns the y coordinate of the turtle's location
void	hide() hides this turtle
void	home() moves the turtle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

סוגים של שרותי מופע:

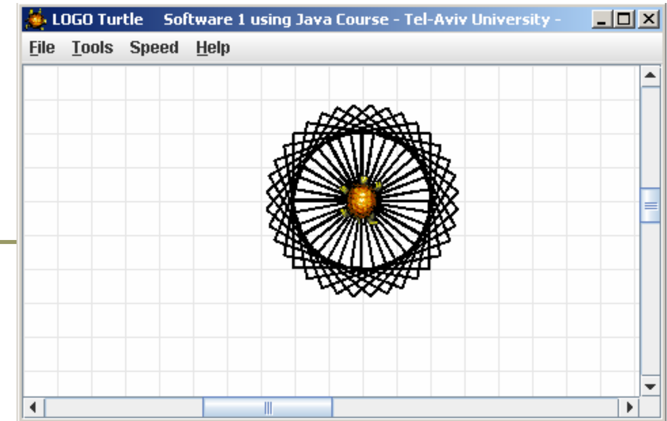
1. **שאילתות** (queries) –
   - שרותים שיש להם ערך מוחזר
   - בדרך כלל לא משנים את מצב העצם)
   - בשיעור הבא נדון בסוגים שונים של שאילתות
2. **פקודות** (commands) –
   - שרותים ללא ערך מוחזר
   - בדרך כלל משנים את מצב העצם שעליו הם פועלים

דוגמת שימוש

```java
public class TurleClient {

    public static void main(String[] args) {
        Turtle leonardo = new Turtle();

        if(!leonardo.isTailDown())
            leonardo.tailDown();

        leonardo.moveForward(50);
        leonardo.turnRight(90);

        leonardo.moveForward(50);
        leonardo.turnRight(90);

        leonardo.moveForward(50);
        leonardo.turnRight(90);

        leonardo.moveForward(50);
        leonardo.turnRight(90);
    }
}
```



עוד דוגמת שימוש




```
public class TurleClient {

    public static void main(String[] args) {
        Turtle leonardo = new Turtle();
        leonardo.tailDown();
        drawSquarePattern(leonardo, 50, 10);
    }

    public static void drawSquare(Turtle t, int size) {
        for (int i = 0; i < 4; i++) {
            t.moveForward(size);
            t.turnRight(90);
        }
    }

    public static void drawSquarePattern(Turtle t, int size, int angle) {
        for (int i = 0; i < 360/angle; i++) {
            drawSquare(t, size);
            t.turnRight(angle);
        }
    }
}
```

"לאונרדו יודע..."

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

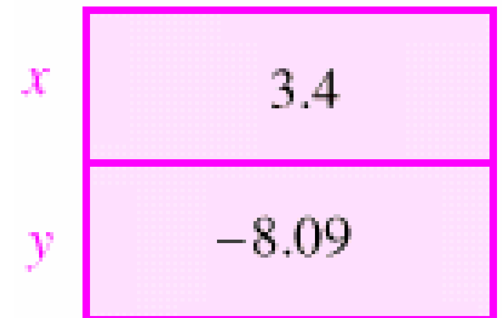
כתיבת מחלקה (כטיפוס נתונים)

- ברמה **הטכנית**, כתיבת מחלקות המייצגות טיפוסים **דומה** לכתיבת מחלקות המייצגות ספריות
- השרותים והמשתנים מוגדרים ללא המציין `static` ויקראו **שרותי מופע** ו- **שדות מופע**
- כל אחד מהעצמים שיווצר מהמחלקה יכיל שדות מופע משלו, שערכיהם אינם תלויים בשדות המופע של עצמים אחרים
- כל אחד משרותי המופע יופעל על **עצם מסוים**. אין צורך לציין עצם זה בשורת הארגומנטים, ושמו תמיד `this`

POINT Class

```
public class POINT {
    private double x;
    private double y;

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

    public double getX(){
      return this.x;
    }

    // More Mothods...
}
```

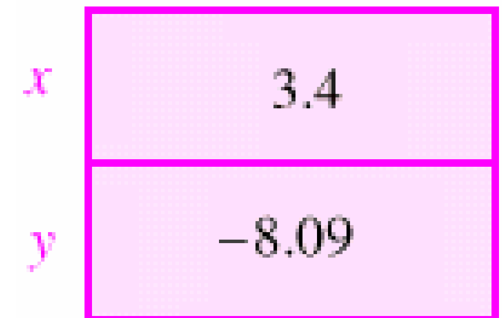
כאשר פונים לשדה מופע או שרות מופע ניתן להשתמש בהפניה this הנוצרת אוטומטית ע"י הקומפיילר של Java



POINT Class

```
public class POINT {
    private double x;
    private double y;

    public void setX(double x){
      this.x = x;
    }

    public void setY(double y){
      this.y = y;
    }

    // More Mothods...
}
```

הדבר שימושי אם שם ארגומנט
לשרות זהה לשם של שדה מופע



Simple Book

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

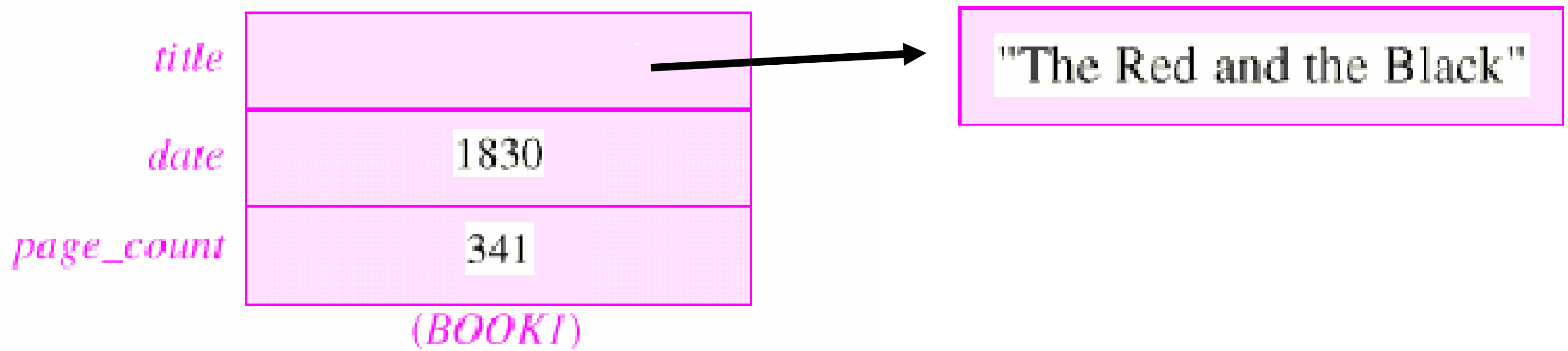
title	"The Red and the Black"
date	1830
page_count	341

(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)

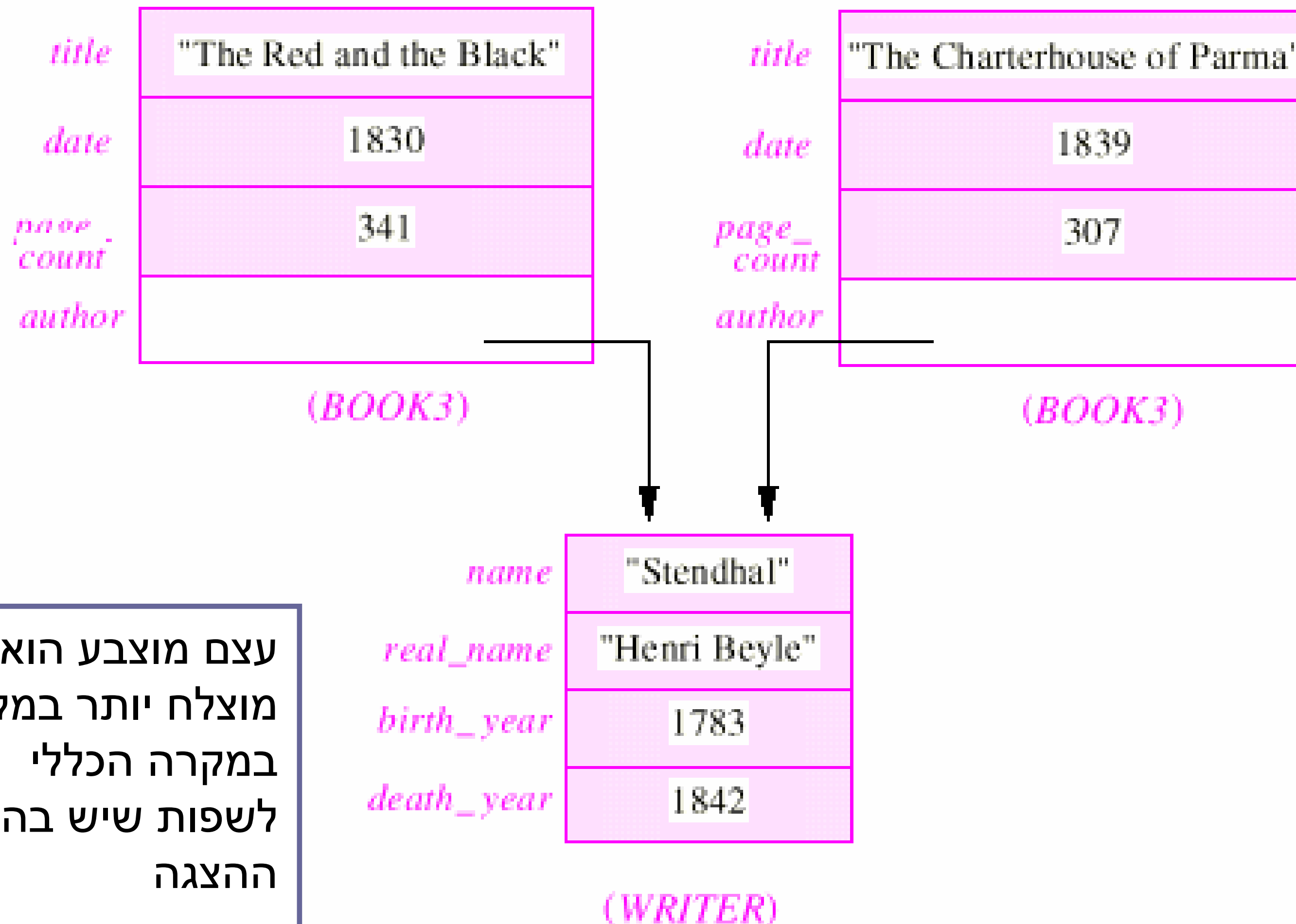
title	"The Red and the Black"
date	1830
page_count	341
	name: "Stendhal" real_name: "Henri Beyle" birth_year: 1783 death_year: 1842 (WRITER)

(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

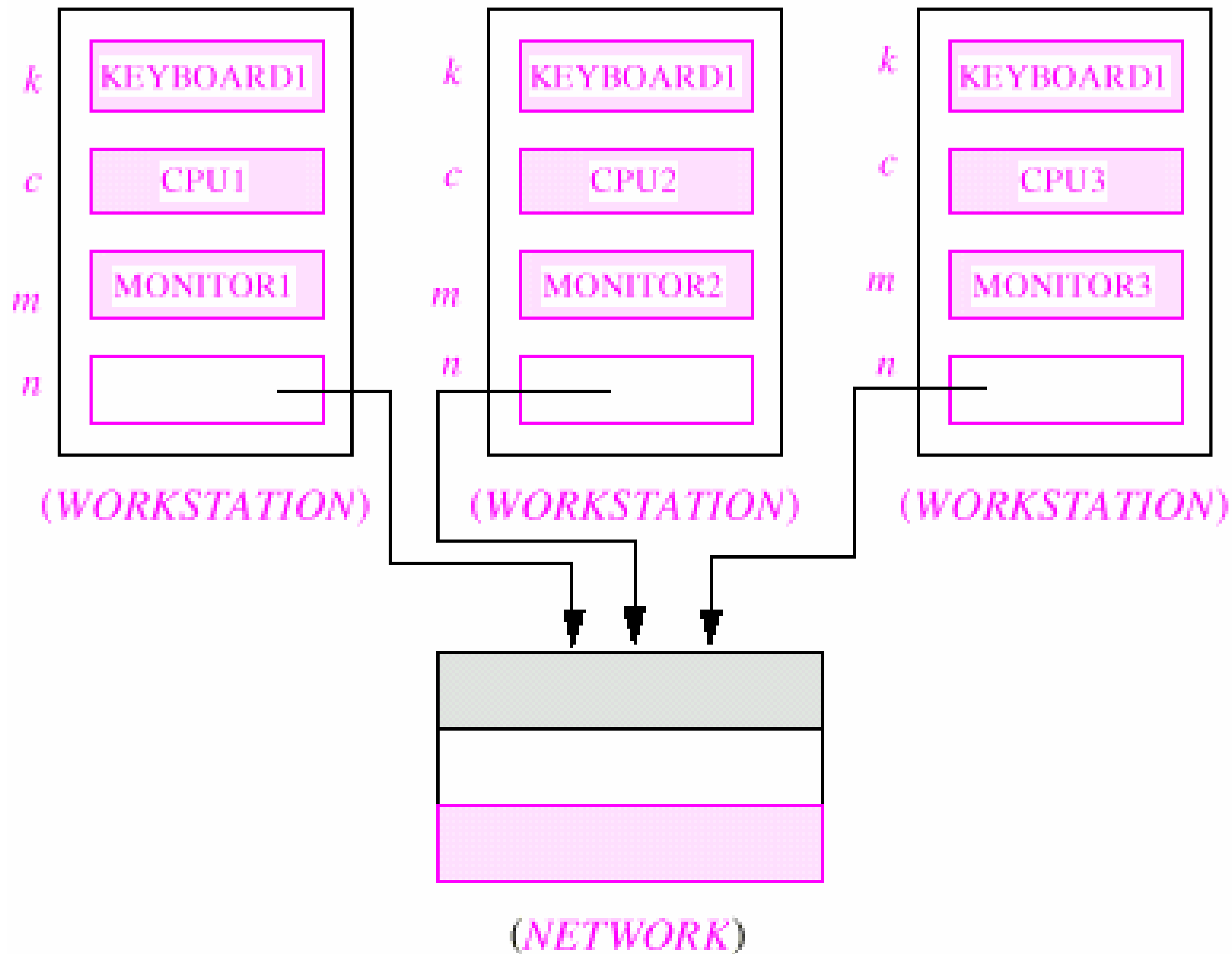


עצם מוצבע הוא כנראה רעיון מוצלח יותר במקרה זה, אולם במקרה הכללי יש יתרונות לשפות שיש בהן שתי צורות הצגה

יתרונות העצם המוכל

- **יעילות**
  - גישה לשדות מוכלים שלא דרך dereference של מצביע
- **מודל טוב יותר** – בהתאם למה שברצוננו לבטא
  - מצביע למחלקה S פירושו שהלקוח "יודע על" S
  - עצם מוכל מעיד על כך שהלקוח <u>מכיל</u> S
  - בפרט, הכלה מרמזת על אי-שיתוף
- **תמיכה אחידה בטיפוסים פרימיטיבים**
  - עצמים מכילים את הטיפוסים היסודיים עצמם ולא מצביע אליהם

אובייקטים מורכבים – מודל טוב יותר



הכלה או הצבעה?

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

לפני שרצים לקודד

- קיימות כמה גישות לפיתוח של קוד בד בבד עם המפרט שלו (specification)
- קיימת חפיפה מסוימת בין הגישות, אולם כל אחת מדגישה פן מסוים של הסתכלות על המערכת
- פרט לציון החוזה של כל שרות (פונקציה) ושל המחלקה כולה בעזרת טענות בולאניות (**D**esign **by** **C**ontract- DbC) נגדיר לטיפוס הנתונים **מצב מופשט ופונקצית הפשטה**
- לפני שנראה את פרטי השיטה נציין **גישה אחרת** הטוענת כי תחילה יש להגדיר ADT (**A**bstract **D**ata **T**ype) – טיפוס נתונים מופשט, וממנו לגזור טענות DbC (**D**esign **by** **C**ontract)

הגדרת ADT

- בגישה זו, נתאר את טיפוס הנתונים **כישות מתמטית** בעזרת:
  - ציון **הטיפוסים** שהוא בנוי מעליהם
  - **פונקציות** (אולי חלקיות) אשר מקבלות את הטיפוס כפרמטר או מחזירות אותו כערך
  - **תנאי קדם** – לציון חלקיות הפונקציות
  - **אקסיומות** – ביטויים לוגיים

- המוטיבציה לגישה זו היא שבהינתן תאור מתמטי ניתן יחסית בקלות להוכיח (מתמטית!) כי הוא **שלם ועקבי**. בעזרת גזירה נכונה ושיטתית של **חוזה** מתוך התאור המתמטי ניתן יהיה לכתוב מימוש שיהיה עקבי עם החוזה ולכן נכונותו מובטחת (עד כדי שלמות החוזה)

הגדרת ADT

- יתרונות
  - מתווה דרך שיטתית לכתיבת תוכנה נכונה
  - בתוך המתודולוגיה שלוב גם מנגנון אכיפת הנכונות

- חסרונות
  - את הכתיבה מקדים שלב של הוכחה מתמטית
  - שלמות - התאור בדרך כלל אינו מספיק
  - מנגנון אכיפה בעזרת ביטויים בולאנים של שפת התכנות לא יכול לבטא תכונות מסוימות
  - ההוכחה עבור מערכות גדולות קשה

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

```java
package il.ac.tau.cs.software1.lec4;

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 */
    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 שטחי ואינו מבטא את מהות הפעולות

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

```java
package il.ac.tau.cs.software1.lec4;

/** @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 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

- ננסח 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 או מאפיינים של **המצב המופשט**
- 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
- מסובך? דווקא פשוט. פשטני.

```
package il.ac.tau.cs.software1.lec4;

/**  @abst (i1, i2, ... , in) or () for the empty stack */
public class StackOfInts {

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

    /**  @abst $ret == i1   */
    public int top(){

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

    /** @abst AF(this) == (i2, i3, ... , in) */
    public void pop()

    /** @abst AF(this) == (x, i1, ... , in) */
    public void push(int x)

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

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

מימוש אפשרי ל StackOfInts

```
package il.ac.tau.cs.software1.lec4;

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];
    }
```

מימוש אפשרי ל StackOfInts (המשך)

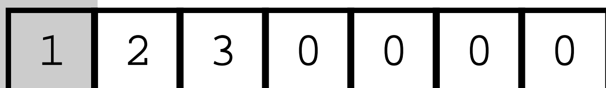
```
public int top(){
   return rep[count];
}

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

public void pop(){
   count--;
}

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

מימוש אפשרי ל StackOfInts (המשך 2)

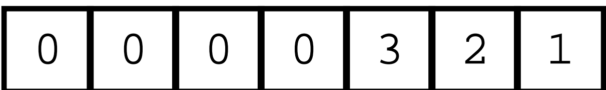
```
public void push(int x){
    if (count == rep.length - 1)
        enlargeRep();
    count++;
    rep[count] = x;
}

/** 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

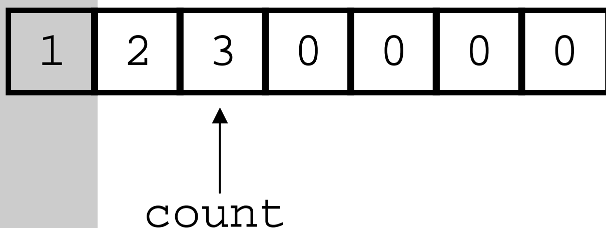


- יכולנו לנקוט גישה אחרת:
  - למלא את האברים מהמקום האחרון לראשון ולרוקן אותם ע"י הגדלת count

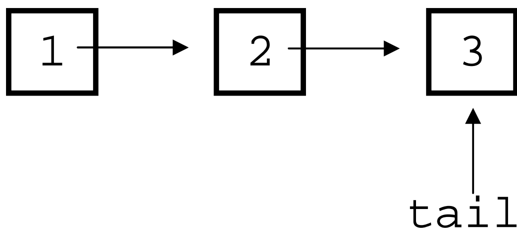


מימוש חלופי ל `StackOfInts`

- 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

- כבררת 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

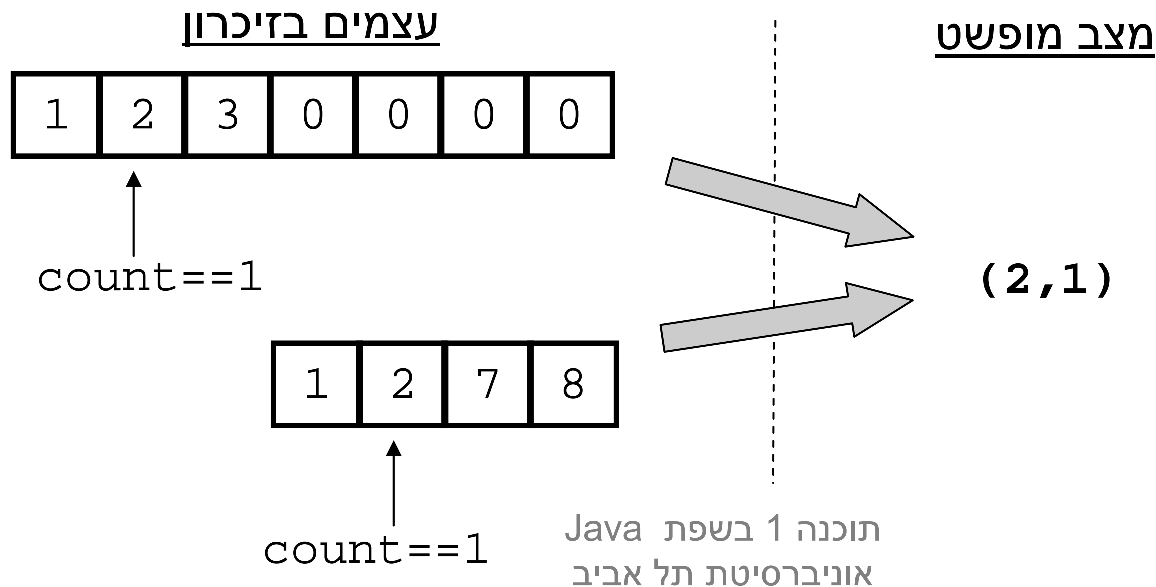
- ראינו כי קיימות דרכים רבות לייצג (לממש) מחלקה
- בחירת הייצוג נקרא **שלב העיצוב** או **שלב התיכון** של המחלקה (design phase)
- לאחר שבחרנו ייצוג למחלקה אנו צריכים להיות עקביים במימוש כדי שהמימוש יהיה תואם למפרט
- לצורך כך עלינו לנסח **פונקצית הפשטה, AF**, הממפה מימוש קונקרטי (ייצוג בזיכרון התוכנית,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

פונקצית ההפשטה ל StackOfInts

$$AF(this) \equiv (x_1, \ldots, x_n) \quad s.t.: \forall i = 1..n : x_i = rep[i-1],$$
$$n = count + 1$$

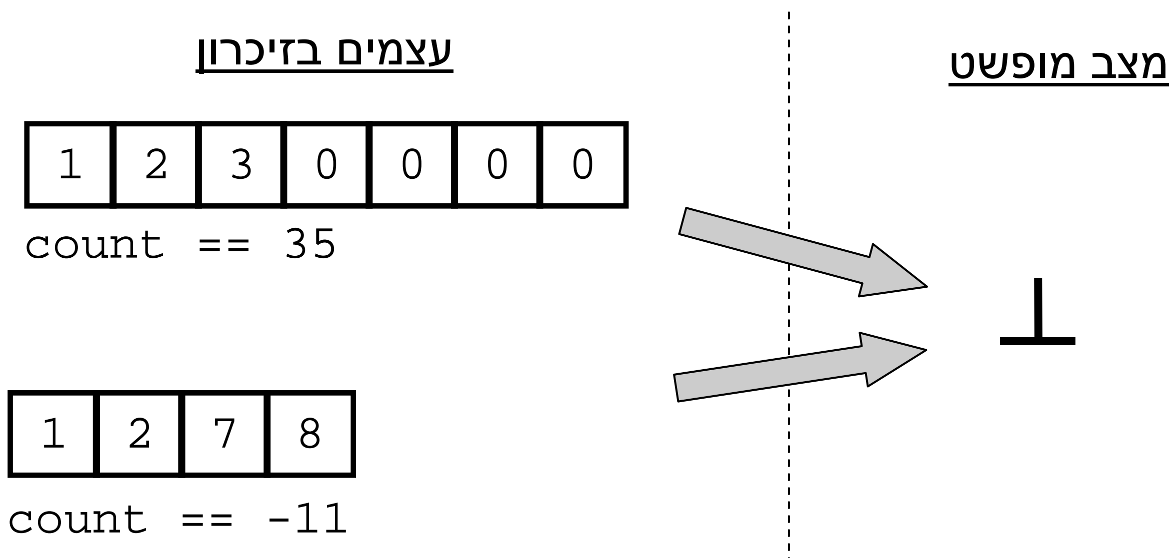
פונקצית ההפשטה אינה חד-חד ערכית

- 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

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



מִשְׁתָמֶר 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 top() == rep[count]
 *  @imp_inv isEmpty() == (count==-1)
 */
public class StackOfInts {
```

הוכחת נכונות של מחלקה

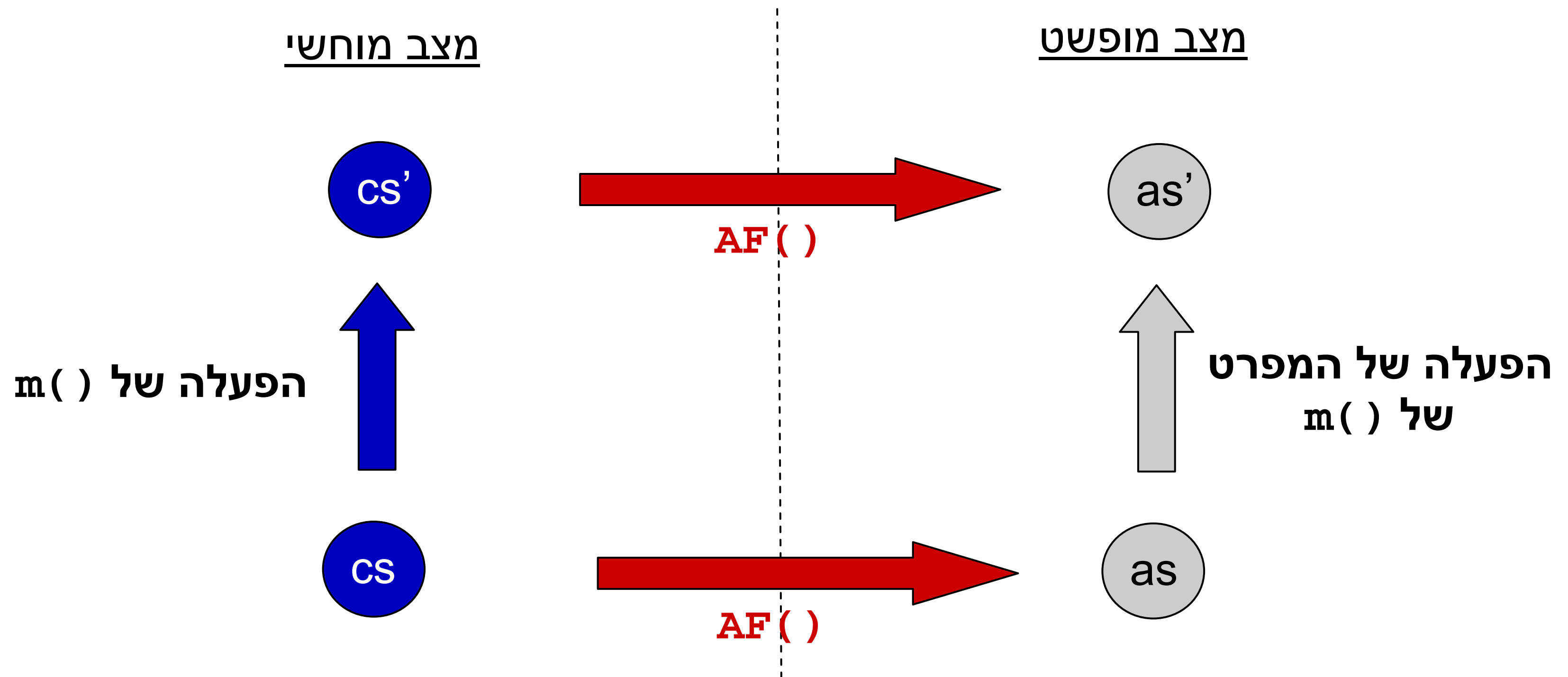
- אולם לא מספיק להראות כי השרותים משרים על העצמים ערכים **חוקיים**, צריך גם להראות כי כל השרותים **עושים מה שהם צריכים** לעשות
  - כלומר מימוש השרותים עקבי עם ההפשטה שנבחרה

- 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:



בנאים

- פונקצית הבנאי נקראת 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

העמסת בנאים

- 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 ecpectedCapacity){
       count = -1;
       rep = new int[ecpectedCapacity];
    }
```

- 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*2];
    }
```

- בשפת Java השימוש ב **this(*args*)** אם קיים, חייב להופיע בשורה הראשונה של הבנאי או מיד אחרי משפט **super** (יוסבר בהמשך הקורס)