תוכנה 1 בשפת Java

שיעור מספר 4: "חוזים"

ליאור וולף

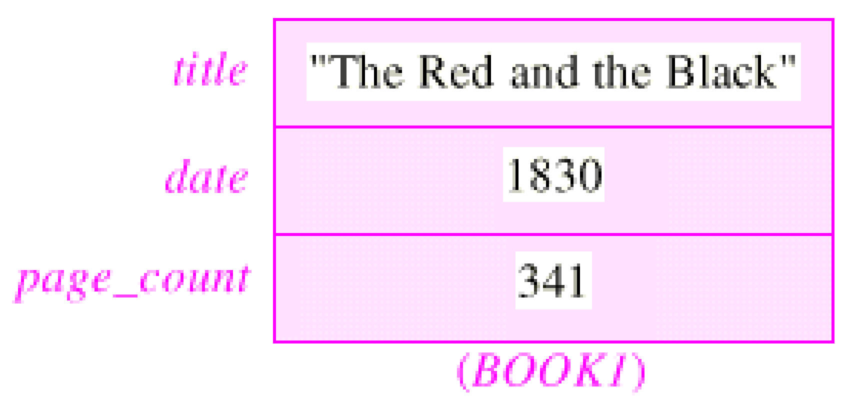
מתי שומרת

בית הספר למדעי המחשב

אוניברסיטת תל אביב

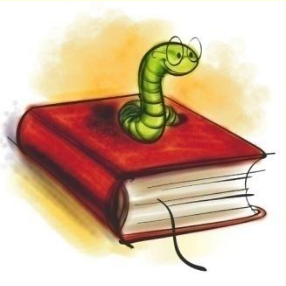
עוד על עצמים ותמונת הזיכרון

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

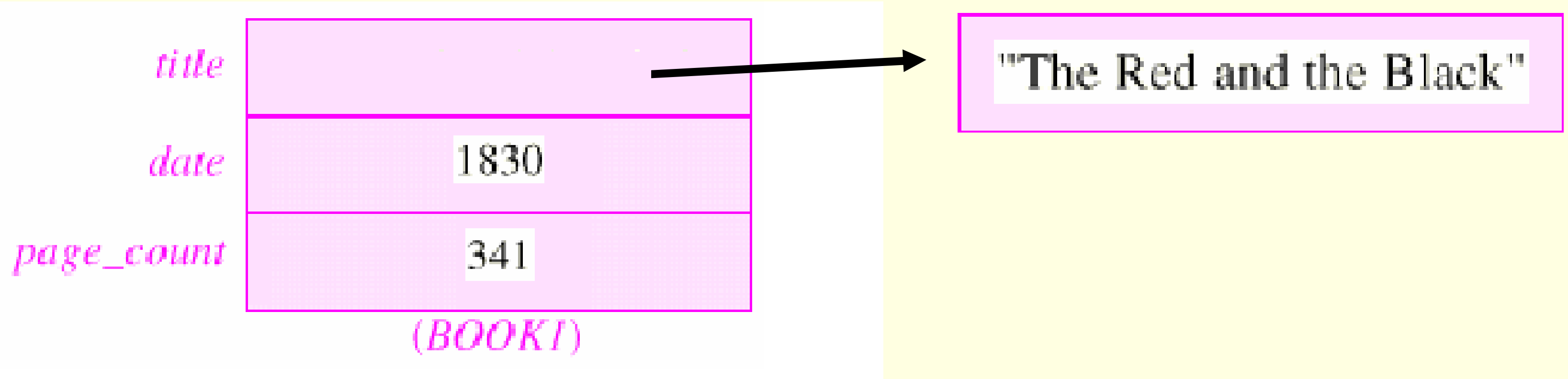


<u>התרשים פשטני</u> –
מחרוזת היא עצם ולכן
השדה title מכיל רק
הפנייה אליו

Simple Book

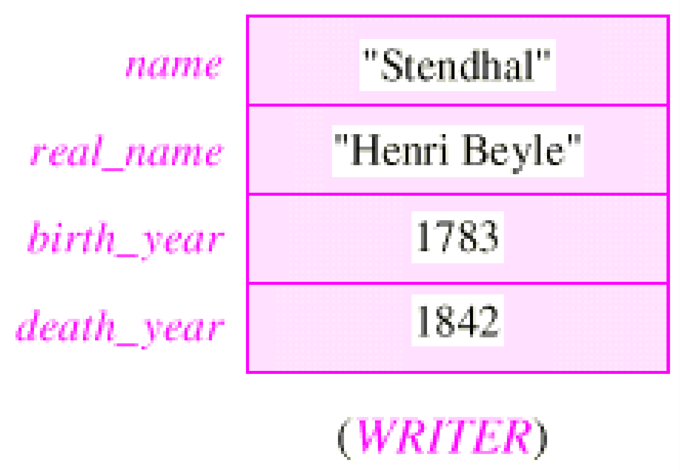


```
public class BOOK1 {
    private String title;
    private int date;
   private int page_count;
}
```



Writer Class

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



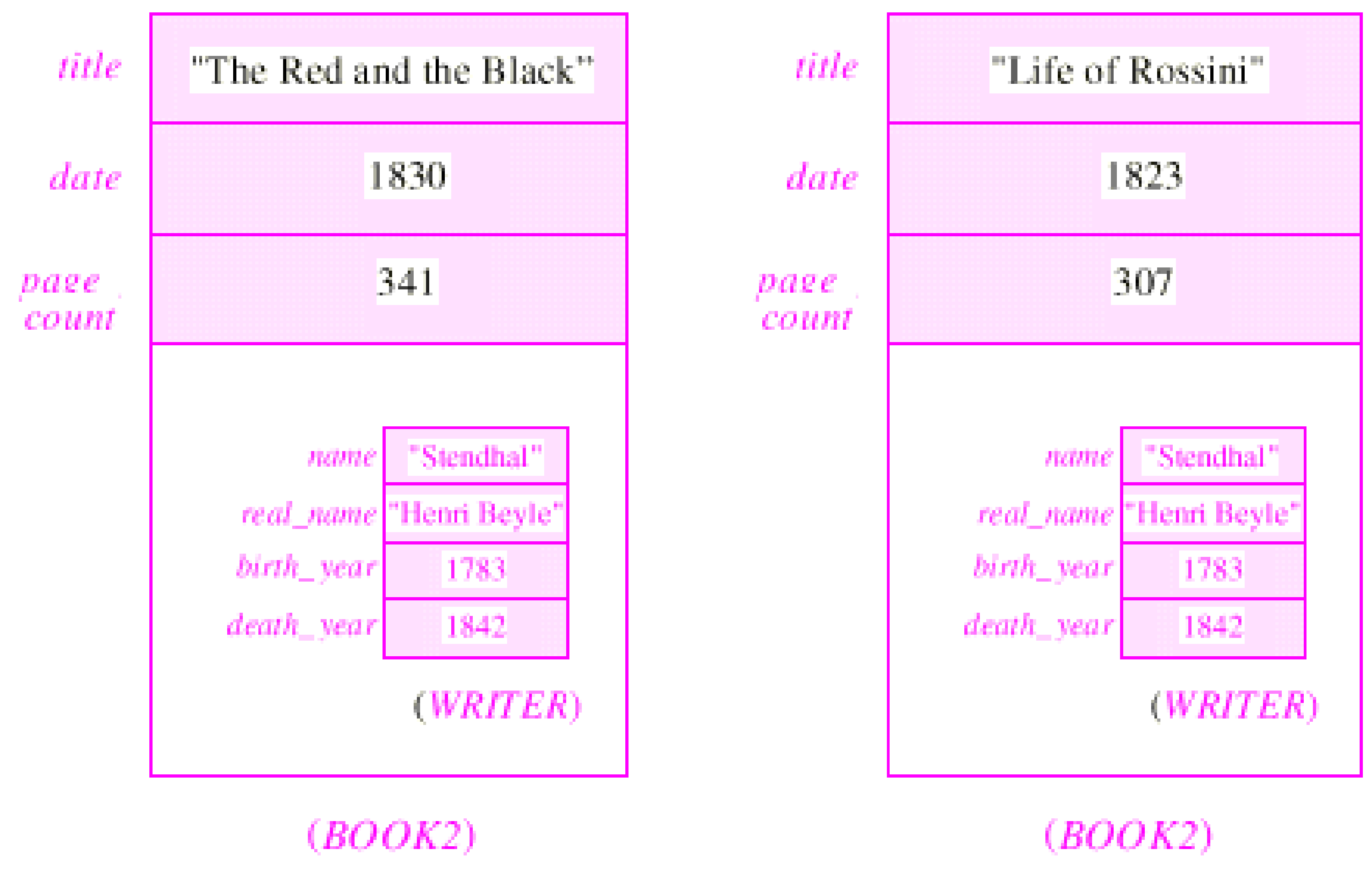
עצמים המתייחסים לעצמים

- איך נבטא את הקשר שבין ספר ומחברו?

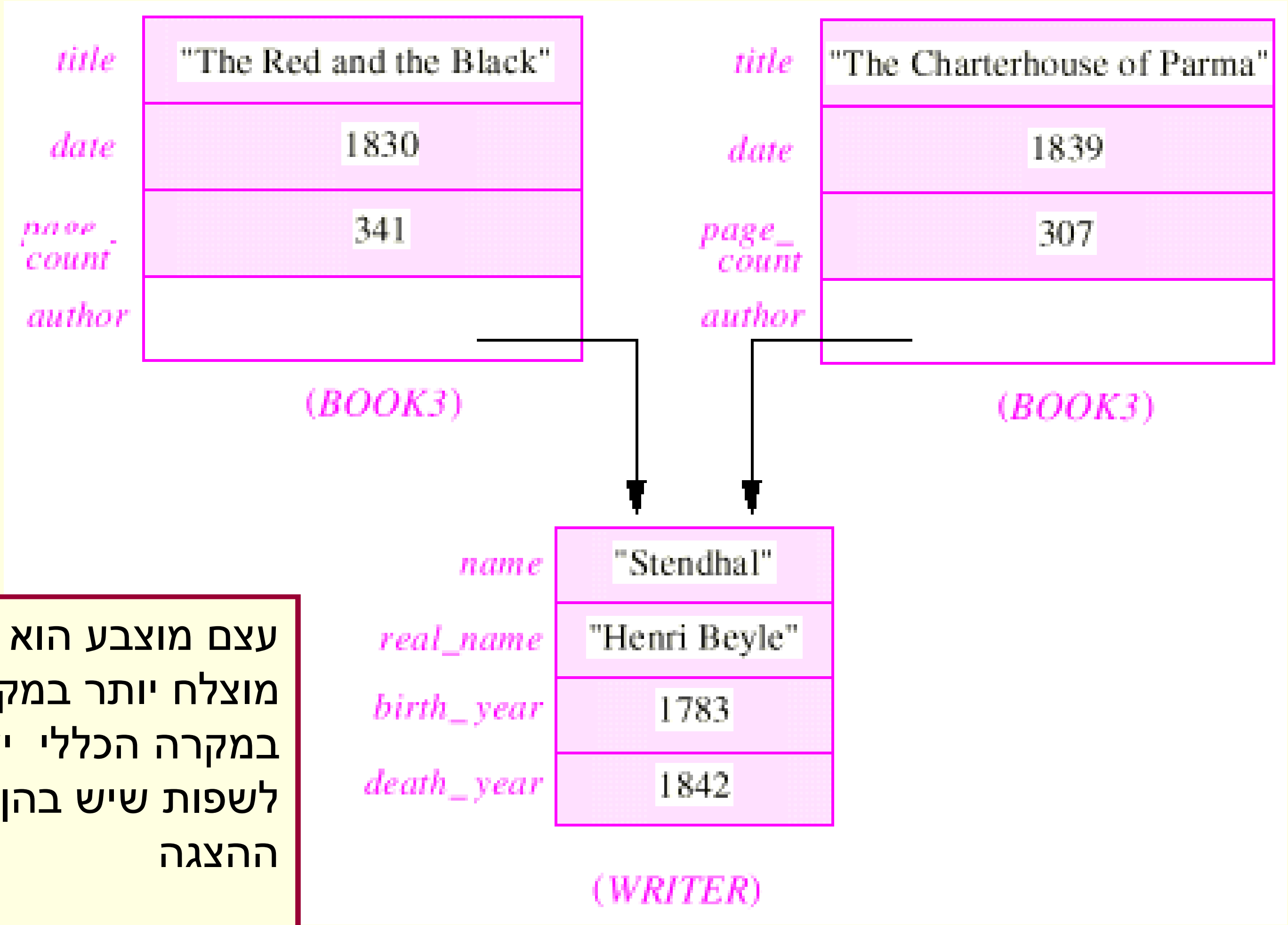
```java
public class BOOK3 {
   private String title;
   private int date;
   private int page_count;
   private Writer author;
}
```

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

עצם מוכל (לא ב- Java)



עצם מוצבע



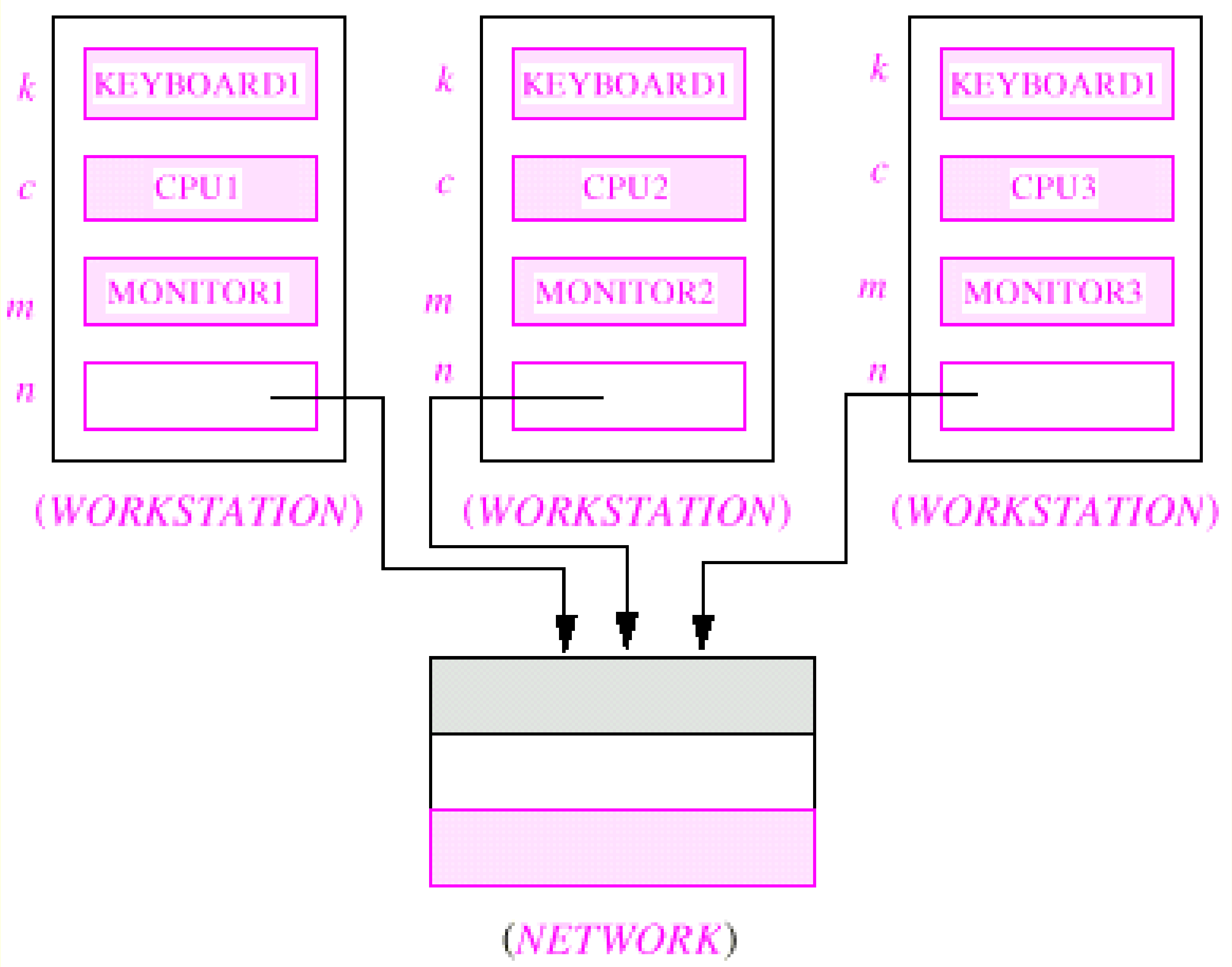
עצם מוצבע הוא כנראה רעיון מוצלח יותר במקרה זה, אולם במקרה הכללי יש יתרונות לשפות שיש בהן שתי צורות ההצגה

יתרונות העצם המוכל

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

- **תמיכה אחידה בטיפוסים 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

לקוח וספק במערכת תוכנה

- **ספק** (supplier) – הוא מי שקוראים לו (לפעמים נקרא גם שרת, server)
- **לקוח** (client) הוא מי שקרא לספק או מי שמשתמש בו (לפעמים נקרא גם משתמש, user). דוגמא:

```
public static void do_something(){
   // doing...
}

public static void main(String [] args) {
   do_something();
}
```

- בדוגמא זו הפונקציה **main** היא **לקוחה** של הפונקציה **do_something()**
- **do_something** היא **ספקית** של **main**

לקוח וספק במערכת תוכנה

- הספק והלקוח עשויים להיכתב בזמנים שונים, במקומות שונים וע"י אנשים שונים ואז כמובן לא יופיעו באותו קובץ (באותה מחלקה)

```
public static void do_something(){
   // doing...
}
```

Supplier.java

```
public static void main(String [] args) {
   do_something();
}
```

Client.java

- חלק נכבד בתעשיית התוכנה עוסק בכתיבת **ספריות** – מחלקות המכילות אוסף שרותים שימושיים בנושא מסוים
- כותב הספרייה נתפס כספק שרותים בתחום (domain) מסוים

פערי הבנה

- חתימה אינה מספיקה, מכיוון שהספק והלקוח אינם רק שני רכיבי תוכנה נפרדים אלא גם לפעמים נכתבים ע"י מתכנתים שונים עשויים להיות פערי הבנה לגבי תפקוד שרות מסוים

- הפערים נובעים ממגבלות השפה הטבעית, פערי תרבות, הבדלי אינטואיציות, ידע מוקדם ומקושי יסודי של תיאור מלא ושיטתי של עולם הבעיה

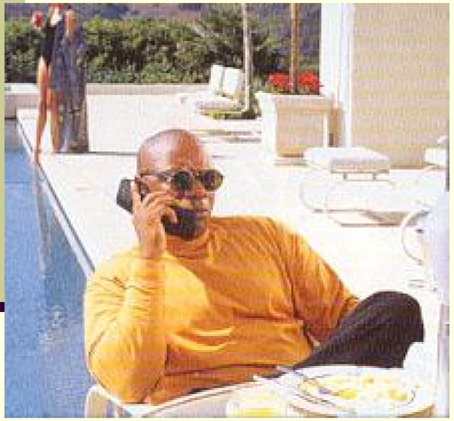
- לדוגמא: נתבונן בשרות **divide** המקבל שני מספרים ומחזיר את המנה שלהם:

```
public static int divide(int numerator, int denominator)
{...}
```

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

"Let us speak of the unspeakable"

- אך מה יוחזר עבור הארגומנטים 7 ו- 2 ?
  - האם הפונקציה מעגלת למעלה?
  - מעגלת למטה?
  - ועבור ערכים שליליים?
  - אולי היא מעגלת לפי השלם הקרוב?
- ואולי השימוש בפונקציה **אסור** בעבור מספרים שאינם מתחלקים ללא שארית?
- מה יקרה אם המכנה הוא אפס?
  - האם נקבל ערך מיוחד השקול לאינסוף?
  - האם קיים הבדל בין אינסוף ומינוס אינסוף?
- ואולי השימוש בפונקציה **אסור** כאשר המכנה הוא אפס?
- מה קורה בעקבות שימוש **אסור** בפונקציה?
  - האם התוכנית **תעוף**?
  - האם מוחזר **ערך שגיאה**? אם כן, איזה?
  - האם קיים משתנה או מנגנון שבאמצעותו ניתן לעקוב אחרי שגיאות שארעו בתוכנית?

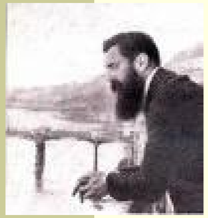


יותר מדי קצוות פתוחים...

- אין בהכרח תשובה **נכונה** לגבי השאלות על הצורה שבה על divide לפעול
- ואולם יש לציין במפורש:
  - מה היו **ההנחות** שביצע כותב הפונקציה
    - במקרה זה הנחות על הארגומנטים (האם הם מתחלקים, אפס במכנה וכו')
  - מהי **התנהגות** הפונקציה במקרים השונים
    - בהתאם לכל המקרים שנכללו בהנחות
- פרוט ההנחות וההתנהגויות השונות מכונה **החוזה** של הפונקציה
- ממש כשם שבעולם העסקים נחתמים חוזים בין ספקים ולקוחות
  - קבלן ודיירים, מוכר וקונים, מלון ואורחים וכו'...

עיצוב על פי חוזה (design by contract)

- בשפת Java אין תחביר מיוחד כחלק מהשפה לציון החוזה, ואולם אנחנו נתבסס על תחביר המקובל במספר כלי תכנות
- נציין בהערות התיעוד שמעל כל פונקציה:
  - **תנאי קדם (precondition)** – מהן **ההנחות** של כותב הפונקציה לגבי הדרך התקינה להשתמש בה
  - **תנאי בתר (תנאי אחר, postcondition) – מה עושה הפונקציה**, בכל אחד מהשימושים התקינים שלה
- נשתדל לתאר את תנאי הקדם ותנאי הבתר במונחים של ביטויים בולאנים חוקיים ככל שניתן (לא תמיד ניתן)
- שימוש בביטויים בולאנים חוקיים:
  - מדויק יותר
  - יאפשר לנו בעתיד **לאכוף** את החוזה בעזרת כלי חיצוני



חוזה אפשרי ל- divide

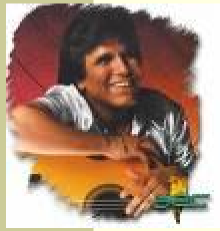
```
/**
 * @pre denominator != 0 ,
 *      “Can’t divide by zero"
 *
 * @post Math.abs($ret * denominator) <= Math.abs(numerator) ,
 *       "always truncates the fraction"
 *
 *  @post (($ret * denominator) + (numerator % denominator)) == numerator,
 *        "regular divide"
 */
public static int divide(int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

- התחביר מבוסס על כלי בשם Jose
- לפעמים החוזה ארוך יותר מגוף הפונקציה

חוזה אפשרי **אחר** ל- divide

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

תנאי קדם סובלניים מסבכים את מימוש הפונקציה – כפי שמתבטא בחוזה

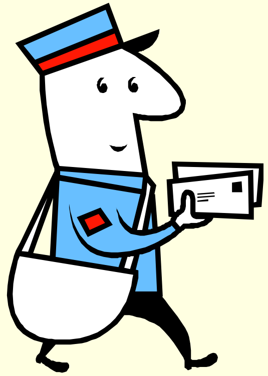


החוזה והמצב

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

הדואר בא היום

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

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

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



שרות לעולם לא יבדוק
את תנאי הקדם שלו

- שרות לעולם לא יבדוק את תנאי הקדם שלו
- גם לא "ליתר ביטחון"
- אם שרות בודק תנאי קדם ופועל לפי תוצאת הבדיקה, אזי יש לו התנהגות מוגדרת היטב עבור אותו תנאי – כלומר הוא אינו תנאי קדם עוד

- אי הבדיקה מאפשרת כתיבת מודולים "סובלניים" שיעטפו קריאות למודולים שאינם מניחים דבר על הקלט שלהם
- כך נפריד את בדיקות התקינות מהלוגיקה העסקית (business logic) כלומר ממה שהפונקציה עושה באמת
- גישת תיכון ע"פ חוזה סותרת גישה בשם "תכנות מתגונן" (defensive programming) שעיקריה לבדוק תמיד הכל

חלוקת אחריות

- אבל מה אם הלקוח שכח לבדוק?
- זו הבעיה שלו!

- החוזה מגדיר במדויק אחריות ואשמה, זכויות וחובות:
  - הלקוח – חייב למלא אחר תנאי הקדם לפני הקריאה לפונקציה (אחרת הספק לא מחויב לדבר)
  - הספק – מתחייב למילוי כל תנאי האחר אם תנאי הקדם התקיים

- הצד השני של המטבע – לאחר קריאה לשרות אין צורך לבדוק שהשרות בוצע.
- ואם הוא לא בוצע? יש לנו את מי להאשים...

דוגמא

```java
/**
 * @param a Am array sorted in ascending order
 * @param x a number to be searched in a
 * @return the first occurrence of x in a, or -1 if not
 *         exists
 *
 * @pre "a is sorted in ascending order"
 */
public static int searchSorted(int [] a, int x)
```

- האם עליה לבדוק את תנאי הקדם?
- כמובן שלא, בדיקה זו עשויה להיות איטית יותר מאשר ביצוע החיפוש עצמו
- ונניח שהיתה בודקת, מה היה עליה לעשות במקרה שהמערך אינו ממוין?
  - להחזיר ‎-1 ?
  - למיין את המערך?
  - לחפש במערך הלא ממוין?
- על `searchSorted` לא לבדוק את תנאי הקדם. אם לקוח יפר אותו היא עלולה להחזיר ערך שגוי או אפילו לא להסתיים אבל זו כבר לא אשמתה...

חיזוק תנאי האחר

- אם תנאי הקדם לא מתקיים, לשירות מותר שלא לקיים את תנאי האחר כשהוא מסיים; קריאה לשירות כאשר תנאי הקדם שלו לא מתקיים מהווה תקלה שמעידה על פגם בתוכנית
- אבל גם אם תנאי הקדם לא מתקיים, **מותר** לשירות לפעול ולקיים את תנאי האחר
- לשירות מותר גם לייצר כאשר הוא מסיים מצב הרבה יותר ספציפי מזה המתואר בתנאי האחר; תנאי האחר לא חייב לתאר **בדיוק** את המצב שייווצר אלא מצב כללי יותר (תנאי חלש יותר)
  - למשל, שירות המתחייב לביצוע חישוב בדיוק של ε כלשהו יכול בפועל להחזיר חישוב בדיוק של ε /2

דע מה אתה מבקש

- מי מונע מאיתנו לעשות שטויות?
- אף אחד

- קיימים כלי תוכנה אשר מחוללים קוד אוטומטי, שיכול לאכוף את קיום החוזה בזמן ריצה ולדווח על כך
- השימוש בהם עדיין לא נפוץ
- אולם, לציון החוזה (אפילו כהערה!) חשיבות מתודולוגית נכבדה בתהליך תכנון ופיתוח מערכות תוכנה גדולות

החוזה והקומפיילר

- יש הבטים מסויימים ביחס שבין ספק ללקוח שהם באחריותו של הקומפיילר
  - למשל: הספק לא צריך לציין **בחוזה** שהוא מצפה ל-2 ארגומנטים מטיפוס **`int`**, מכיוון שחתימת המתודה והקומפיילר מבטיחים זאת

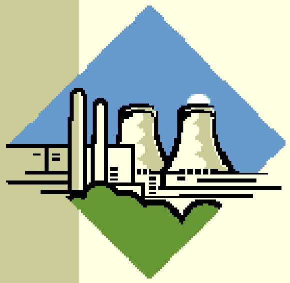
- ספק לא יודע באילו הקשרים (context) יקראו לו
  - מי יקרא לו, עם אילו ארגומנטים, מה יהיה ערכם של משתנים גלובלים מסוימים ברגע הקריאה
  - רבים מההקשרים יתבררו רק בזמן ריצה

- הקומפיילר יודע לחשב רק מאפיינים סטטיים (כגון התאמת טיפוסים)

- לכן תנאי הקדם של החוזה יתמקדו בהקשרי הקריאה לשרות
  - ערכי הארגומנטים
  - ערכי משתנים אחרים ("המצב של התוכנית")

טענות על המצב

- האם התוכנה שכתבנו **נכונה**?
- איך נגדיר **נכונות**?
- **משתמר** (שמורה, invariant) – הוא ביטוי בולאני שערכו נכון 'תמיד'
- נוכיח כי התוכנה שלנו נכונה ע"י כך שנגדיר עבורה משתמר, **ונוכיח** שערכו true בכל רגע נתון
- להוכחה פורמלית (בעזרת לוגיקה) יש חשיבות מכיוון שהיא מנטרלת את **הדו משמעיות** של השפה הטבעית וכן היא לא מניחה דבר על **אופן השימוש** בתוכנה



זהו אינו "דיון אקדמי"

- להוכחת נכונות של תוכנה חשיבות גדולה במגוון רחב של יישומים

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

```
plutoniumLevel < CRITICAL_MASS_THRESHOLD
```

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

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

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

הוכחת נכונות של טענה

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

```
/** @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"

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

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

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

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

דוגמא נגדית

```
public class CounterExample {

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

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

נראות פרטית (private visibility)

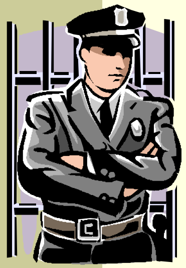
- הגדרת משתנה או שרות כ **private** מאפשרים גישה אליו רק מתוך המחלקה שבה הוגדר:

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

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

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

```java
public class CounterExample {

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

הסתרת מידע והכמסה

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



גישה מבוקרת

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

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

  private static int counter;

  public static int getCounter() {
      return counter;
  }

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

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

גישה מבוקרת

- הלקוחות ניגשים למונה דרך המתודה שמספק להם הספק

```
public class CounterExample {

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

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

משתמר הייצוג

- ראינו שימוש בחוזה של מחלקה כדי לבטא בצורה מפורשת את גבולות האחריות עם לקוחות המחלקה
- אולם, ניתן להשתמש במתודולוגיה של "עיצוב ע"פ חוזה" גם "לצורכי פנים"
- כשם שהחוזה מבטא הנחות והתנהגות בצורה פורמלית יותר מאשר הערות בשפה טבעית, כך ניתן להוסיף טענות בולאניות לגבי היבטים של המימוש
- כדי שלא לבלבל את הלקוחות עם משתמר המכיל ביטויים שאינם מוכרים להם, נגדיר **משתמר ייצוג** המיועד לספקי המחלקה בלבד

משתמר הייצוג

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

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

   private static int counter;

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

תנאי בתר ייצוגי

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

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

- אבל לא בתנאי קדם של מתודות ציבוריות
- מדוע?

מתודות עזר

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

נראות ברמת החבילה (package friendly)

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

הוכחת החוזה

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

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 (DbC - **D**esign **b**y **C**ontract) נגדיר לטיפוס הנתונים **מצב מופשט ופונקצית הפשטה**

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

```java
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 הנתונים המופשט 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 if 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 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 true if stack is empty
     *  @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

- המתודה ()count אינה חלק מהקונספט של מחסנית
- גם בעזרתה לא ניתן לתאר את המהות שבפעולות
- 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.

```
/**  @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 וערך מוחזר במונחים מופשטים

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

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

- עבור שאילתות, התיאור מציין מהו הערך יוחזר

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

  - שאילתא אינה משנה את המצב

- 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

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)

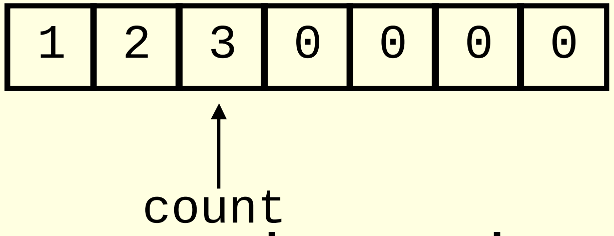
```
public int top(){
   return rep[count];
}

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

public void pop(){
   count--;
}

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

מימוש אפשרי ל StackOfInts (המשך 2)

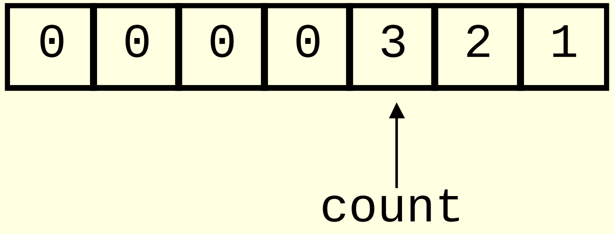
```
public void push(int x){
  if (count == rep.length - 1)
     enlargeRep();
  count++;
  rep[count] = x;
}

/** 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

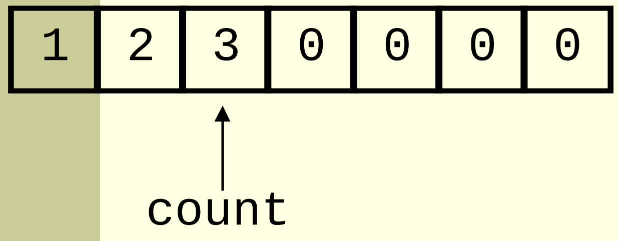


- יכולנו לנקוט גישה אחרת:
  - למלא את האברים מהמקום האחרון לראשון ולרוקן אותם ע"י הגדלת count

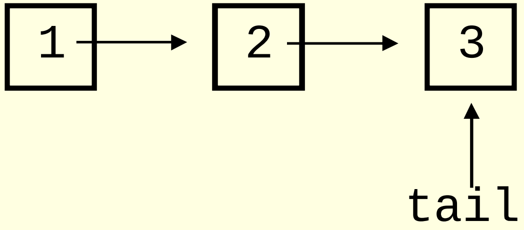


מימוש חלופי ל StackOfInts

- 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

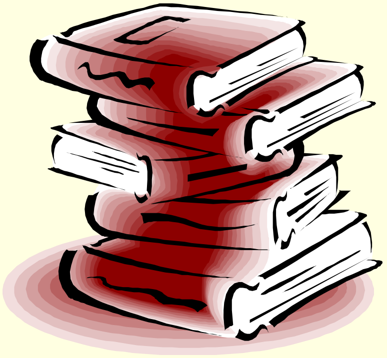
- **לכן תמיד נסתיר את הייצוג הפנימי מלקוחותינו**

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

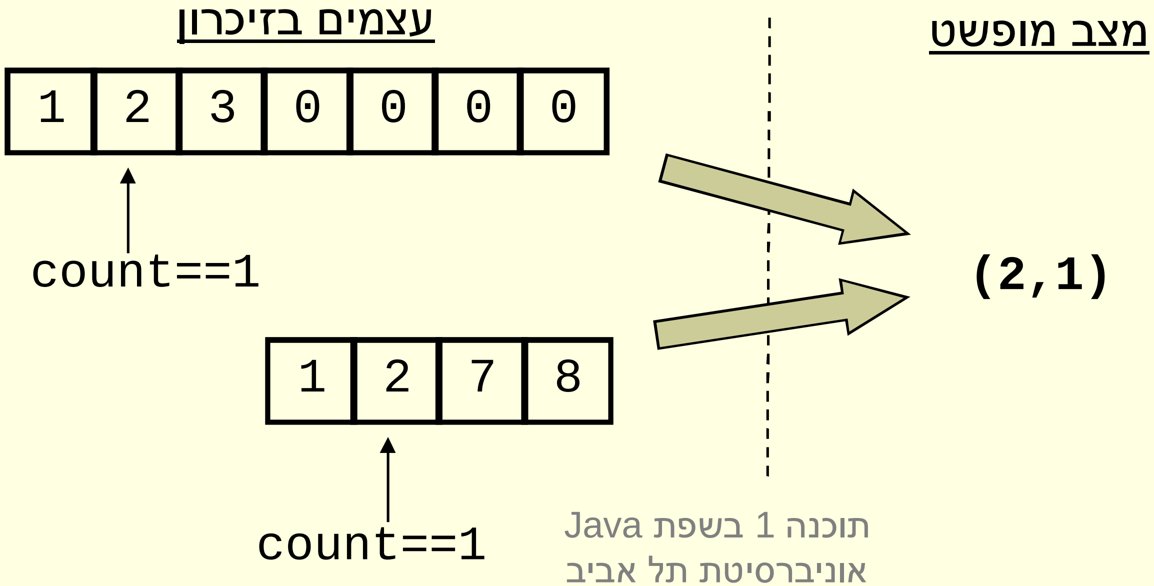
פונקצית הפשטה ל StackOfInts

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



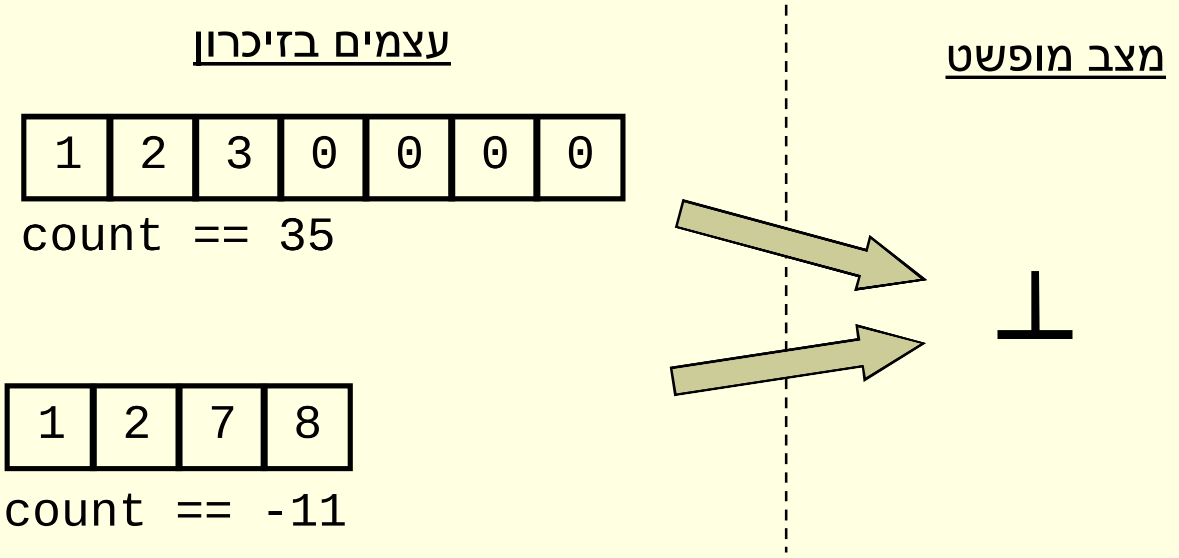
פונקצית ההפשטה אינה חד-חד ערכית

- פונקצית ההפשטה בדרך כלל **אינה חד-חד ערכית** כלומר היא many to one:
  - 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



מִשְׁתָּמֶר הייצוג

- מכיוון שעצם אמור לייצג בכל רגע נתון מצב מופשט כלשהו, צריכים להתקיים אילוצים מסוימים 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 {
```

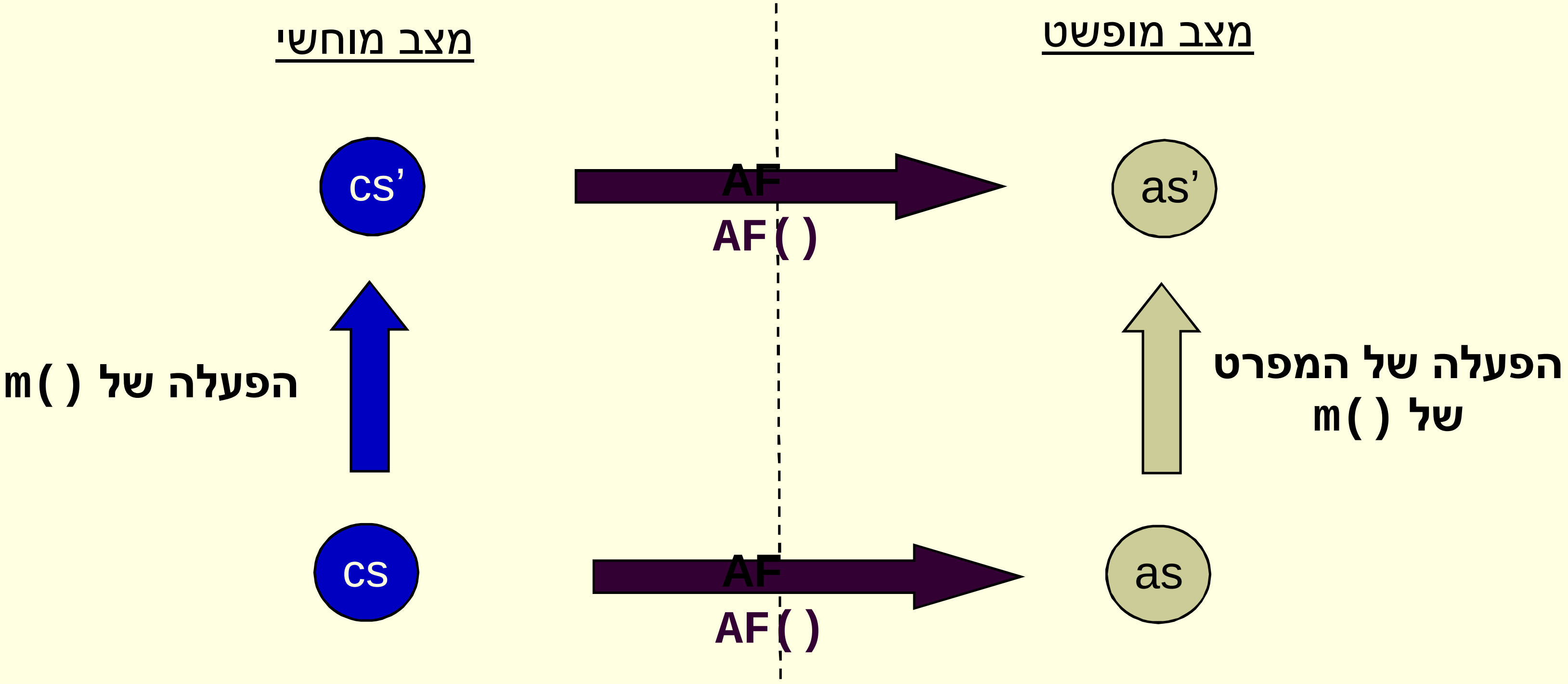
הוכחת נכונות של מחלקה

- 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).Spec_m()
```

נכונות המימוש

- כלומר שני המסלולים בתרשים שקולים:



העמסת בנאים

- 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