תוכנה 1 בשפת Java

שיעור מספר 5: "הייה נכון – נכון תמיד"

אוהד ברזילי

דן הלפרין

בית הספר למדעי המחשב

אוניברסיטת תל אביב

על סדר היום

- חוזים, נכונות והסתרת מידע
- כתיבת מחלקות חדשות לפי מפרט
- מצב מופשט של עצם

- עקרונות תכנות נכונים
- ספריות לעומת יישומים
- ומבוא לעיצוב תוכנה

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()`:

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

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

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

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

"הוכחה"

- **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

```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

```
/** @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
- 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, בהמשך השיעור)
  - 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"

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());
s1.push(1);
System.out.println("s1.top() == " + s1.top());
s1.push(2);
System.out.println("s1.top() == " + s1.top());
s1.pop();
System.out.println("s1.top() == " + s1.top());
System.out.println("isEmpty() == " + s1.isEmpty());
```

- נציג חוזה לטיפוס המחסנית

```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 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) { ... }

}
```

```java
/** @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() { ... }
}
```

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 (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 יוחזר

```
@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)

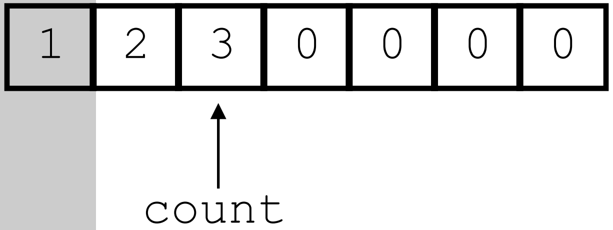
```
public int top(){
    return rep[count];
}

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

public void pop(){
    count--;
}

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

מימוש אפשרי ל StackOfInts (המשך 2)

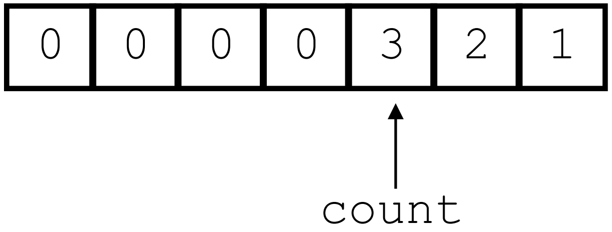
```
public void push(int x){
    if (count == rep.length - 1)
        enlargeRep();
    count++;
    rep[count] = x;
}

/** 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

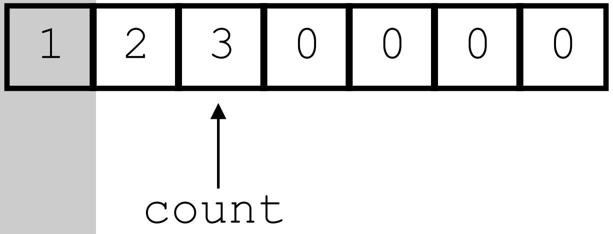


- יכולנו לנקוט גישה אחרת:
  - למלא את האברים מהמקום האחרון לראשון ולרוקן אותם ע"י הגדלת count

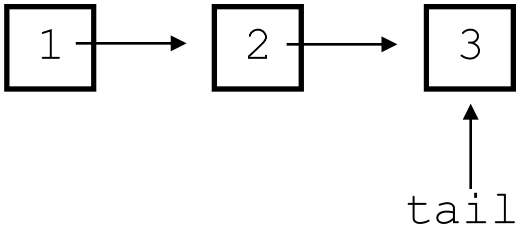


מימוש חלופי ל StackOfInts

- 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
- 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 הפשטה

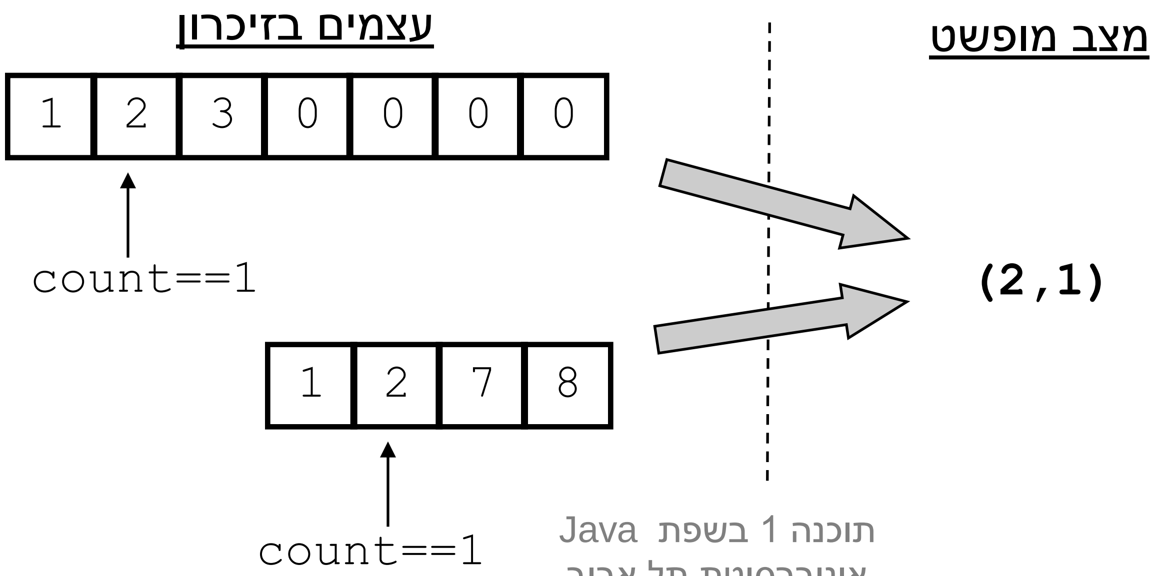
- ראינו כי קיימות דרכים רבות לייצג (לממש) מחלקה
- בחירת הייצוג נקרא **שלב העיצוב** או **שלב התיכון** של המחלקה (design phase)
- לאחר שבחרנו ייצוג למחלקה אנו צריכים להיות עקביים במימוש כדי שהמימוש יהיה תואם למפרט
- לצורך כך עלינו לנסח **פונקצית הפשטה, AF**, הממפה מימוש קונקרטי (ייצוג בזיכרון התוכנית,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

פונקצית ההפשטה ל StackOfInts

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

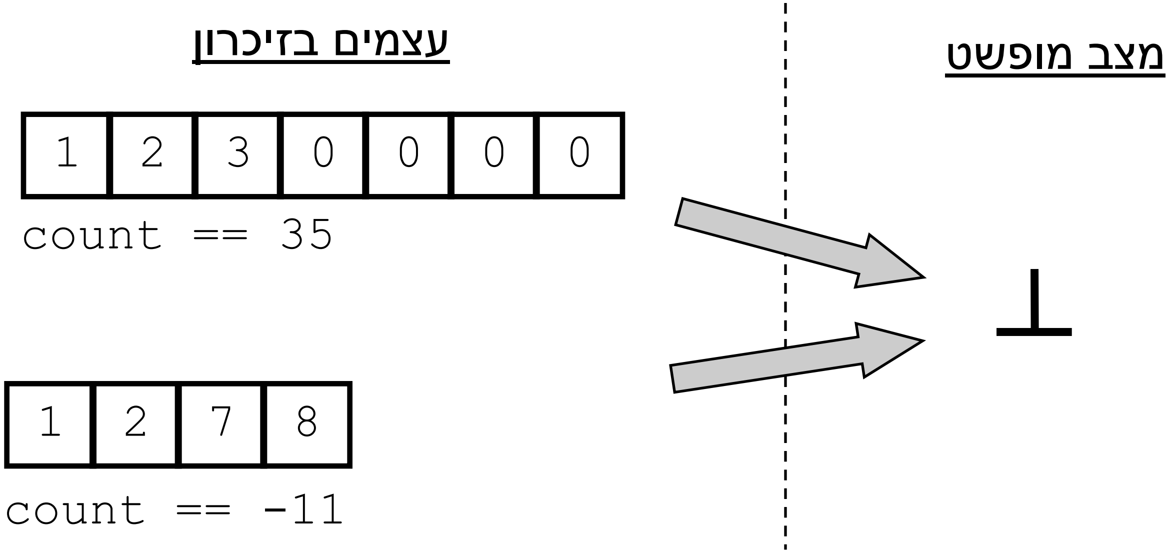
פונקצית ההפשטה אינה חד-חד ערכית

- 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



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

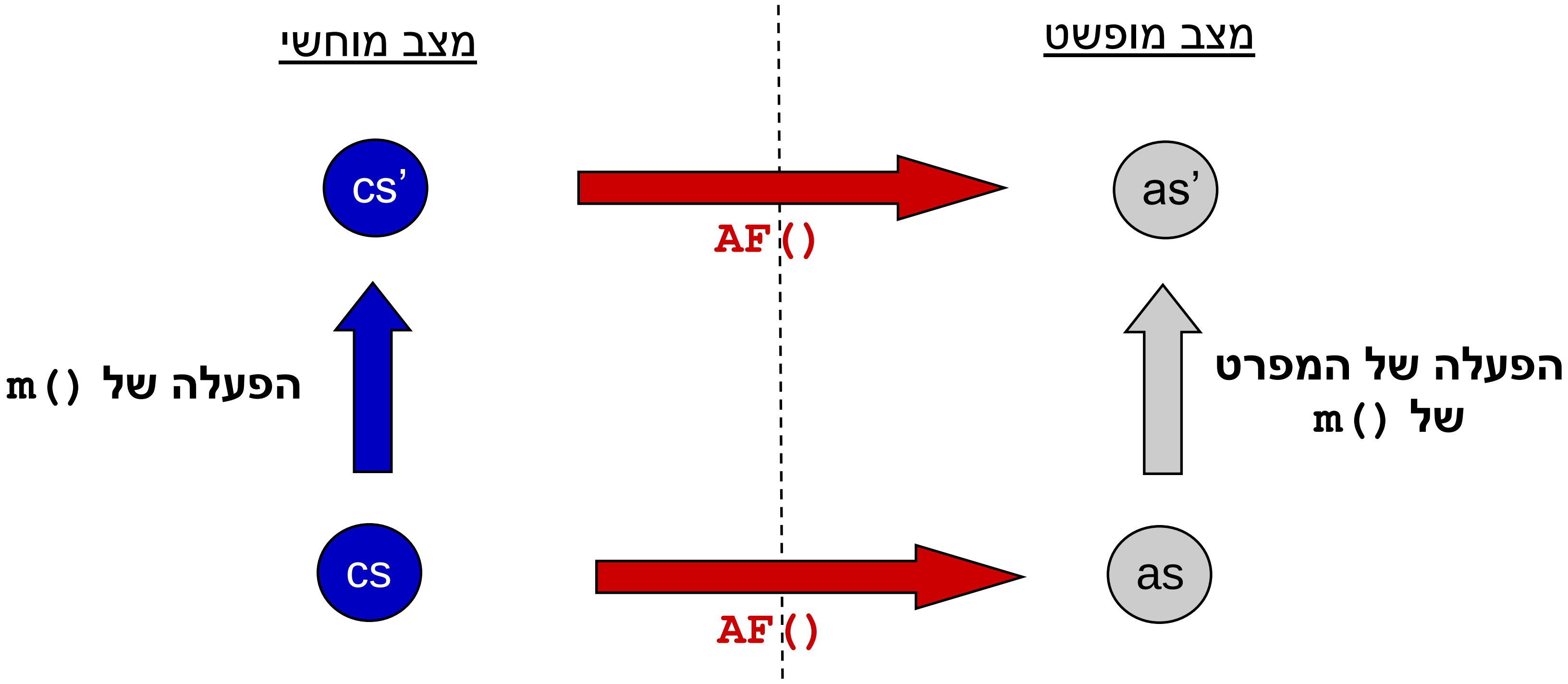
- אולם לא מספיק להראות כי השרותים והבנאים משרים על העצמים ערכים **חוקיים**, צריך גם להראות כי כל השרותים **עושים מה שהם צריכים** לעשות
  - כלומר מימוש השרותים עקבי עם ההפשטה שנבחרה
  - ומתקיים החוזה של כל השרותים והבנאים

- 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

- 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*2];
    }
```

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

תכנות מונחה עצמים

"משבר התוכנה"

- כתיבת תוכנית קטנה אינה משימה קשה
- הקושי בהנדסת תוכנה נעוץ בבניית מערכות תוכנה גדולות (>1,000,000 שורות קוד), ע"י מספר אנשים הדורשות תחזוקה לאורך זמן רב (כמה שנים)
- הגודל כן קובע
- מאז סוף שנות ה-60 של המאה ה-20 ברור לעולם התוכנה, כי בשביל להתגבר על הקושי שבכתיבת מערכות גדולות יש צורך להלביש על שפת התכנות עקרונות פיתוח נכונים אשר ישפרו את היכולת לכתוב מערכות תוכנה מורכבות

תכנות מונחה עצמים

- אחד מאוספי העקרונות האלה מכונה "תכנות מונחה עצמים" (**O**bject **O**riented **P**rogramming) שעיקריו (כפי שיבואו לידי ביטוי בקורס):
  - שימוש חוזר בקוד
  - הפשטה
  - מודולריות
  - תיכון בעזרת חוזים (design by contract)
  - ביצוע מקסימום בדיקות תקינות בזמן קומפילציה
  - קשה בקומפילציה קל בזמן הריצה
  - ניהול זיכרון אוטומטי
- חלק משפות התכנות המודרניות מקדמות עקרונות אלו ע"י הגדרת מבנים בשפה שיתמכו בהם בצורה ישירה או עקיפה
- שפת Java היא שפה כזו

שימוש חוזר בתוכנה

- על מנת לשמור על עלויות תוכנה סבירות, יש לשפר את תפוקת מפתחי התוכנה
- שיפור תפוקה יומית של מתכנת דורש שיפורים משמעותיים בתהליכי הפיתוח, שפות התכנות, וכלי הפיתוח
- בנוסף, ניתן להקטין את עלות הפיתוח ע"י שימוש ברכיבי תוכנה קיימים, שפותחו עבור פרויקט קודם או פותחו במיוחד כתשתית לארגון
- שימוש חוזר בתוכנה כרוך בקשיים רבים, לא כולם טכניים: תסמונת "לא הומצא אצלנו", תשלום עבור תוכנה לפי שורות קוד
- הניסיון מראה שרכיבי תוכנה מונחת עצמים מתאימים לשימוש חוזר יותר מרכיבים פרוצדורליים

מודולריות

- מודולריות פירושה היכולת לפרק מערכת למרכיבים, לבנות מערכת ממרכיבים, להבין כל מודול בפני עצמו
- נחוצה כדי לאפשר הפרדת עניינים בזמן הפיתוח, מעודדת שימוש חוזר ומשפרת קריאות לצורך תחזוקה
- מודולריות טובה כתכונה של מערכת דורשת מודולים בעלי חוזק פנימי גבוה, וצמידות נמוכה
- פונקציה היא מודול. מחלקה היא מודול.
- מתברר שארכיטקטורת מערכת שמבוססת על הנתונים מאפשרת מודולריות טובה יותר מארכיטקטורה שמבוססת על הפונקציונליות
- מכאן היתרון של פיתוח תוכנה מונחה עצמים

Java כשפה מונחית עצמים

- לאחר שכיסינו את רוב היסודות הפרוצדורלים של שפת Java, נקדיש את רוב הקורס למבנים בשפה אשר מקדמים עקרונות תכנות נכונים
- ננסה להבין עבור כל מבנה תחבירי כזה – איך השימוש בו ישפר את איכות הקוד הנוצר

שרותים

- לשימוש בשרותים יש מרכיב מרכזי בבניית מערכות תוכנה גדולות בכמה מישורים:
  - חסכון בשכפול קוד
  - עליה ברמת ההפשטה
  - הגדרת יחסי ספק-לקוח בין כותב השרות והמשתמשים בשרות

שרותים - חסכון בשכפול קוד

- אם קטע קוד מופיע יותר מפעם אחת (copy-paste) יש להפוך אותו לפונקציה (שרות)

- אם הקוד המופיע דומה אבל לא זהה יש לבדוק האם אפשר לבטא את השוני כפרמטר לשרות, או להשתמש בקריאות הדדיות במידת הצורך

- בהקשר זה ראינו בתרגול את תכונת **ההעמסה** (overloading) ב Java. לשתי פונקציות עם אותו השם יש כנראה גם מימוש דומה

שרותים והפשטה

- **גם אם אין חסכון** בשכפול קוד יש חשיבות בהפיכת קוד למתודה
- המתודה מתפקדת כקופסא שחורה המאפשרת לקורא הקוד להבין את הלוגיקה שלו בקלות, ולתחזק אותו ביעילות
  - "מרוב עצים לא רואים את היער"
  - לדוגמא: אין צורך לקרוא את מימוש הפונקציה `sort` כדי להבין מה היא עושה
- שיקולי יעילות (קפיצה נוספת למתודה מאיטה במעט את ריצת הקוד) הם משניים בשיקולי פיתוח מערכות תוכנה גדולות
  - קומפיילרים חכמים, אופטימייזרים ומעבדים חזקים משמעותיים בהרבה

```java
public static void printOwing(double amount) {
    //printBanner
    System.out.println("********************");
    System.out.println("*** Customer Owes **");
    System.out.println("********************");

    //print details
    System.out.println ("name:" + name);
    System.out.println ("amount" + amount);
}
```

שרותים והפשטה
דוגמא

```java
public static void printOwing(double amount) {
    printBanner();
    printDetails(amount);
}

public static void printBanner() {
    System.out.println("********************");
    System.out.println("*** Customer Owes **");
    System.out.println("********************");
}

public static void printDetails(double amount) {
    System.out.println ("name:" + name);
    System.out.println ("amount" + amount);
}
```

שכתוב מבני (refactoring)

- ישנן פעולות של שכתוב קוד שהן כל כך שכיחות עד שהומצא להן שם
  - לדוגמא: הפיכת קטע קוד לשרות שראינו בשקף הקודם נקרא: "חלץ למתודה" (extract method)
- בשנים האחרונות נאסף מספר גדול של פעולות כאלה וקובץ בקטלוג בשם Refactoring. הקטלוג זמין ברשת ובכמה ספרים

http://www.refactoring.com/catalog/index.html

- סביבות פיתוח מודרניות (לרבות Eclipse) מאפשרות **שכתובים אוטומטיים** בלחיצת כפתור
- ביצוע שכתוב בעזרת כלי אוטומטי פותר בעיות רבות של חוסר עקביות העשויות להיווצר כאשר הוא מתבצע ידנית
  - למשל: החלפת שם משתנה בצורה עקבית או חילוץ קטע קוד התלוי במשתנה מקומי

מחלקות והפשטה

- שימוש בשמות משמעותיים עבור מחלקות משרת את אותה המטרה
  - המחלקה כספריה: די ברור מה יש במחלקה **`Math`** או **`Arrays`**
  - המחלקה כטיפוס: אין צורך לדעת את המימוש הפנימי של **`String`** או **`Turtle`** כדי להשתמש בהן
- במהלך הקורס נראה מנגנונים נוספים (מנשקים) שבהם אפילו אין צורך לדעת את **שם המחלקה** כדי להשתמש בה

ספריות ויישומים
(Libraries and Applications)

ספריות

- ספרייה היא מודול תוכנה המכיל אוסף של טיפוסים ושרותים שימושיים
- כותב המחלקה (הספק) אמור לענות על צרכי לקוחותיו, כאשר הוא אינו יודע ורצוי שלא יניח הנחות מרומזות על **הקשרי השימוש** במחלקה שלו
- לדוגמא: מחלקות הספרייה **`String, Date, LinkedList`** מספקות לוגיקה שימושית בהקשרים רבים
- לספרייה אין **`main`**
- אז איך מריצים אותה?
- לא מריצים. משתמשים. לקוחות של הספרייה, אולי ספריות בעצמן, ייצרו ממנה מופעים או ישתמשו בשרותיה

ספריות

- ספרייה אינה מדפיסה למסך
- כדי לתקשר עם לקוחותיה ספרייה יכולה להחזיר ערכים או לשנות ערכי שדות
- לקוחות של הספרייה, אולי ספריות בעצמן, יקבלו ממנה את הערך המוחזר ויחליטו מה לעשות איתו, לפי המידע שברשותן

- ספרייה אינה קולטת קלט מהמשתמש, אלא מקבלת אותו כארגומנטים (או כשדות של העצם שעליו היא פועלת)
- אם נדרש קלט מהמשתמש, לקוחות הספרייה יקלטו אותו ויעבירו לה אותו כארגומנט

יישומים

- **יישום** הוא בעצם שימוש באוסף של ספריות \ מחלקות קיימות וקוד חדש לצורך פתרון **בעיה ידועה**
  - כגון: תוכנית לסנכרון כתוביות, מערכת לניהול ספרייה, מעבד תמלילים, תוכנת דואר
- כותב היישום, בשונה מכותב המחלקה, יודע מה הבעיה שברצוננו לפתור ולכן יכול לממש את היישום בהקשר זה בלבד
  - למשל: הוא יכול להדפיס למסך (כי הוא יודע שיש מסך), הוא יכול לקרוא קלט מהמשתמש או מהרשת, הוא יכול להחליט על טיפול במקרי קצה

יישומים

- כאשר היישום הוא גדול ומורכב (למשל מעבד תמלילים) קיים קושי להחליט עד כמה כללי יהיה הקוד
  - שכן מחלקה אשר נכתבה כחלק ממימוש של מודול מסוים עשויה להתגלות כשימושית גם עבור מודול אחר
  - שימוש חוזר בתוך היישום

- כדי לאפשר שימוש חוזר בתוך היישום יש צורך בראייה כוללת של כל חלקי המערכת ותכנון החלקים המשותפים מראש. שלב זה נקרא גם **עיצוב** או **תיכון**

עיצוב ספריות

- מודול אמור לספק ללקוחותיו את הכלים לעבוד איתו ללא צורך להכיר את מבנהו הפנימי
  - **A**pplication **P**rogramming **I**nterface (API)

- עקרונות ל API מוצלח:
  - פשוט
  - קטן
  - שימושי
  - סימטרי

- דרוש הרבה נסיון כדי לכתוב API מוצלח - לדעת מה הלקוחות רוצים וצריכים

How To Design A Good API and Why it Matters
http://video.google.com/videoplay?docid=-3733345136856180693

עיצוב יישומים

- פירוק פונקציונלי
  - לדוגמא: "המערכת תאפשר **להשאיל** ספר"
  - לדוגמא: "התוכנה תאפשר **לשלוח** דוא"ל"
- פירוק מונחה עצמים
  - לדוגמא: **ספר**, **השאלה**, **שואל**
  - לדוגמא: **הודעה**, **תיבת דואר**, **כתובת**, **איש-קשר**
- בעיצוב תוכנה מודרני מקובל לתכנן מערכות תוכנה על פי **שמות הישויות מעולם הבעיה**, מתוך ההנחה כי המעבר בין העיצוב ובין מימושו כמחלקות טבעי ויעודד שימוש חוזר
- העיצוב מתבצע בדרך כלל במספר רמות הפשטה כדי לאפשר הבנה של הקוד על ידי מספר גורמים. למשל: מתכנתים, מנהלים ואנשי שיווק