תוכנה 1 בשפת Java
שיעור מספר 4: מחלקות ומנשקים

מועבר ע"י
ליאור וולף

בית הספר למדעי המחשב
אוניברסיטת תל אביב

על סדר היום

- שימוש במחלקות קיימות
- כתיבת מחלקות חדשות
- הוכחת נכונות של מחלקות
- מנשקים תחילה

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

- כותב המחלקה לא סיפק 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

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

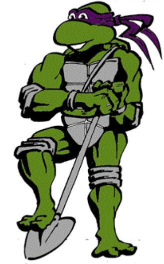
סוגים של שרותי מופע:

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

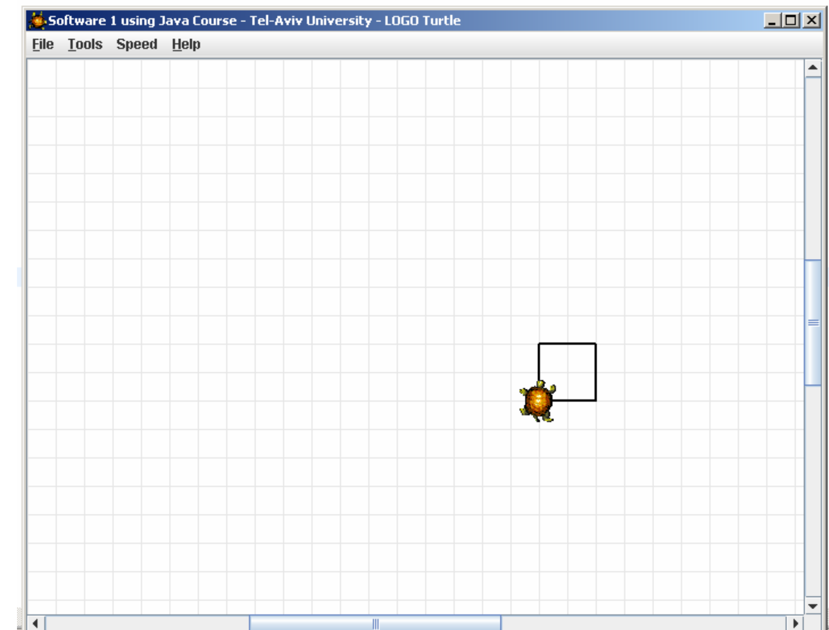
Method Summary

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

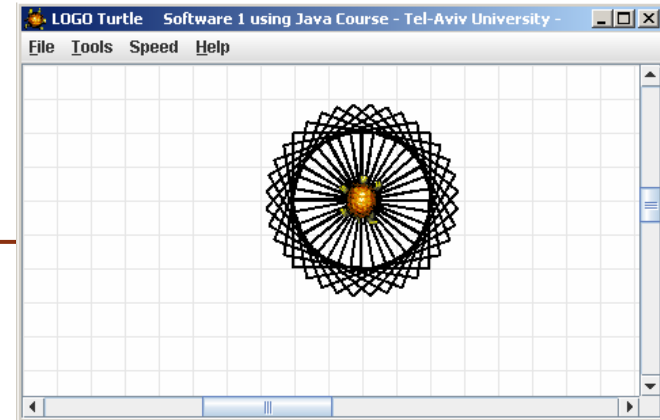
דוגמת שימוש



```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




```java
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)

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

```
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.

```java
/**  @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, 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)

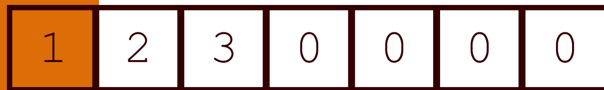
```
public int top(){
    return rep[count];
}

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

public void pop(){
    count--;
}

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

מימוש אפשרי ל StackOfInts (המשך 2)

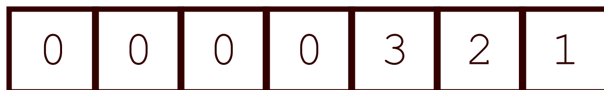
```
public void push(int x){
    if (count == rep.length - 1)
        enlargeRep();
    count++;
    rep[count] = x;
}

/** 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

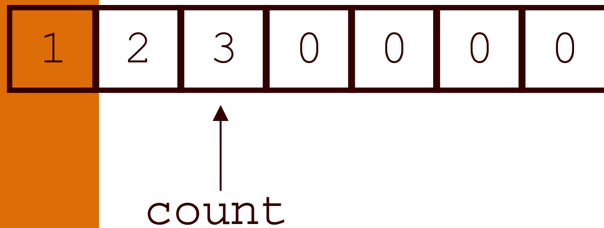


- יכולנו לנקוט גישה אחרת:
  - למלא את האברים מהמקום האחרון לראשון ולרוקן אותם ע"י הגדלת count

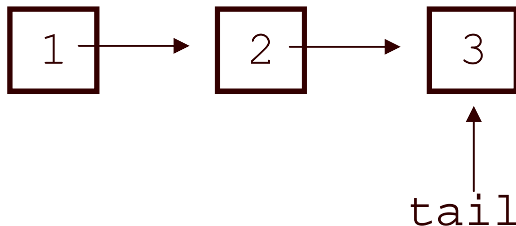


מימוש חלופי ל StackOfInts

- 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**

javadoc ונראות

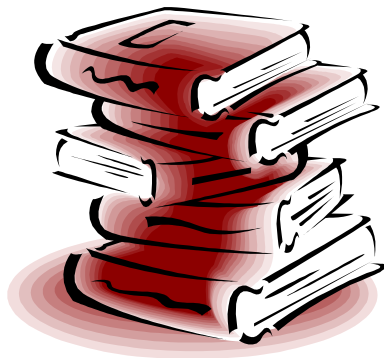
- כלי התיעוד javadoc תומך בדרגות ניראות שונות
- כבררת מחדל, במסמך התיעוד הנוצר אין אזכור של מרכיבי המחלקה הפרטיים (אפילו לא שמם!)
- ניתן להגדיר את דרגת הנראות בעת יצירת התיעוד, וכך להפיק מסמכי תיעוד שונים למפתחי המחלקה וללקוחות המחלקה (אולי מפתחים בעצמם)

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

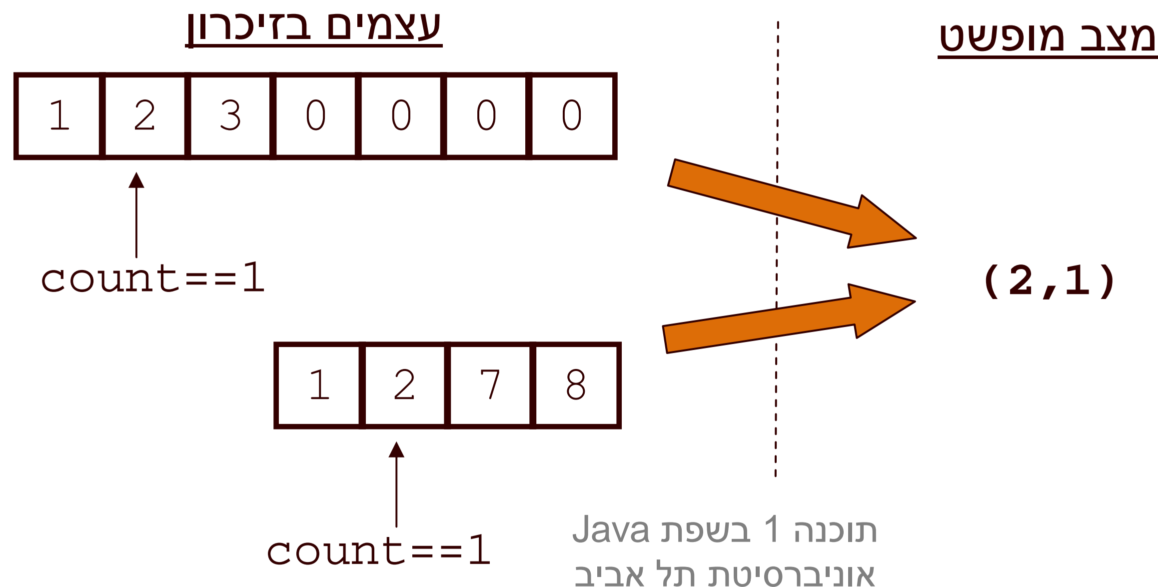
פונקצית הפשטה ל StackOfInts

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



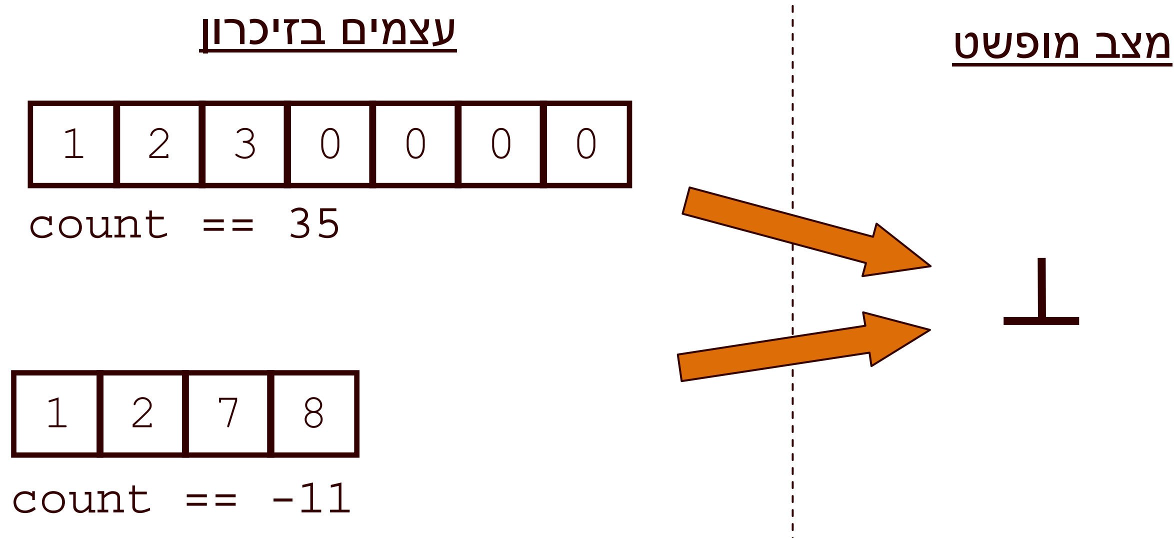
פונקצית ההפשטה אינה חד-חד ערכית

- 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

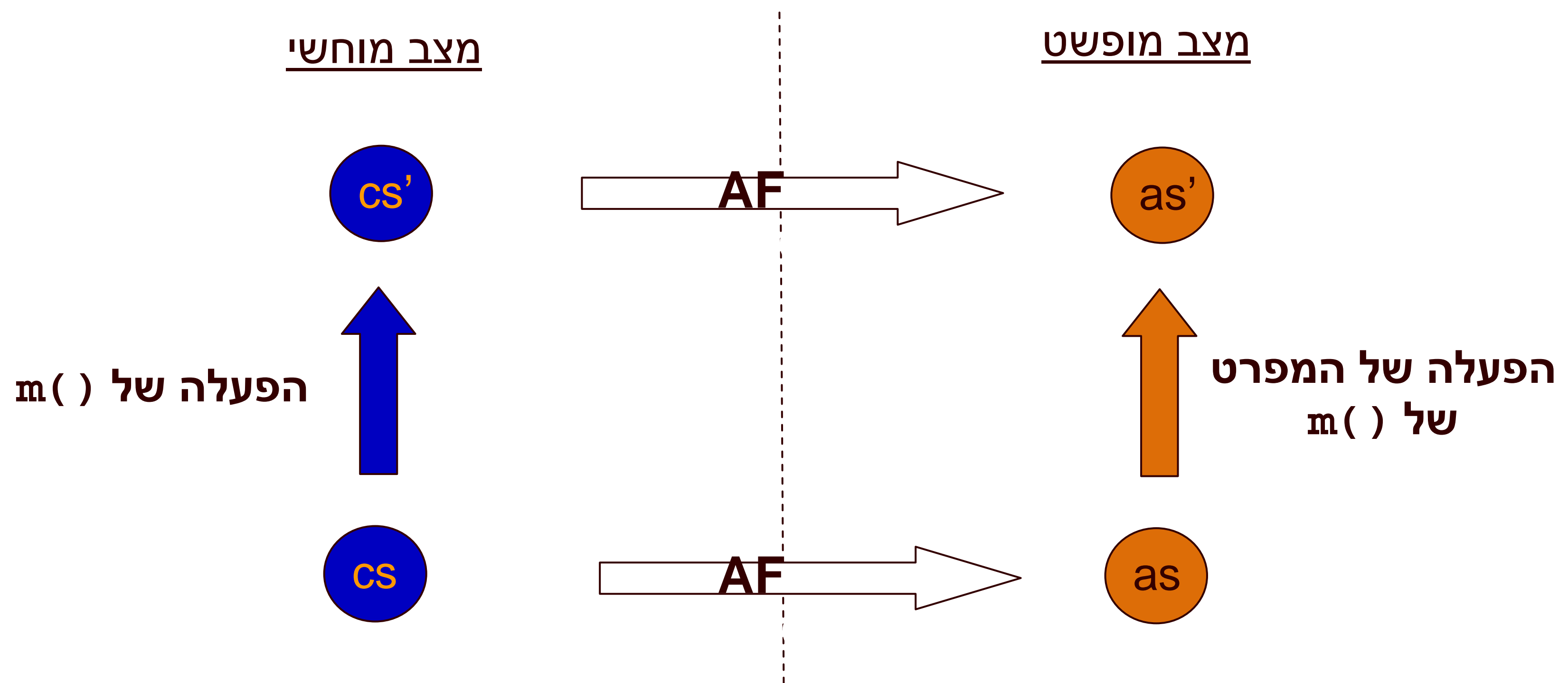
- אולם לא מספיק להראות כי השרותים משרים על העצמים ערכים **חוקיים**, צריך גם להראות כי כל השרותים **עושים מה שהם צריכים** לעשות
  - כלומר מימוש השרותים עקבי עם ההפשטה שנבחרה

- 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

מנשקים תחילה

לקוח וספק במערכת תוכנה - תזכורת

- **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
- יחס בין שרותים או מחלקות (או כל מודול אחר)

לקוח וספק במערכת תוכנה - תזכורת

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

Supplier.java

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

Client.java

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

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

לקוח וספק במערכת תוכנה

- מנקודת מבט של הלקוח קוד המקור של הספק עשוי לא להיות זמין כלל

Supplier.class

```
#@!!!>>??%^#&@
```

Client.java

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

מנשק תחילה

- כדי לתקשר בין הספק והלקוח עליהם להגדיר מנשק (interface, ממשק) ביניהם
- בתהליך פיתוח תוכנה תקין, כתיבת המנשק תעשה בתחילת תהליך הפיתוח
- כל מודול מגדיר מהם השרותים שלהם הוא זקוק ממודולים אחרים ע"י ניסוח מנשק רצוי
- מנשק זה מהווה בסיס לכתיבת הקוד הן בצד הספק, שיממש את הפונקציות הדרושות והן בצד הלקוח, שמשתמש בפונקציות (קורא להן) ללא תלות במימוש שלהן

ראינו יצירת מנשק בעזרת תיעוד

בצד הלקוח

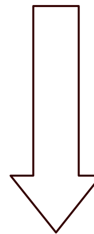
בצד הספק

Supplier.html

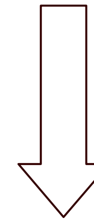
Class Supplier

java.lang.Object
  └ **Supplier**

Method Summary	
static void	do_something() Documentation for do_something goes here...



משתמש בפונקציות לפי המנשק



מממש ע"פ המנשק את הפונקציות

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

Client.java

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

Supplier.java

מנשקים

- מטרת המנשק היא למזער את התלות בין חלקים שונים של המערכת

- מנשקים זעירים יוצרים מערכת:
  - קלה יותר להבנה
  - בעלת הסתרת מידע טובה יותר
  - קלה לתחזוקה ושינויים

מנשקים C ו- Java

- ניתוח והבנה של מערכת תוכנה במונחי ספק-לקוח והמנשקים בינהם היא אבן יסוד בכתיבת תוכנה מודרנית
- בשפת C המנשק מושג ע"י שימוש **בקובצי כותרת** (.h) ואינו מתבטא **בשפת התכנות**, ה pre-processor הוא זה שיוצר אותו, ועל המתכנת לאכוף את עיקביותו
- בשפת Java ניתן להגדיר מנשק ע"י שימוש **בקובצי תיעוד** (בעזרת javadoc) ואולם ניתן לבטא את המנשק גם כרכיב בשפה אשר **המהדר** אוכף את עקביותו
- למתכנתי C:
  - ב- Java אין קובצי כותרת (header files)
  - ב- Java אין צורך להצהיר על פונקציות לפני השימוש בהן

מנשקים (interfaces)

- המנשק הוא מבנה תחבירי בשפה (בדומה למחלקה) המייצג טיפוס נתונים מופשט
  - המנשק מכיל הצהרות על שרותים ציבוריים שיהיו לטיפוס, כלומר הוא מכיל את חתימת השרותים בלבד – ללא מימוש
- בניגוד למחלקה, המנשק לא כולל את המימוש של הטיפוס
- בתהליך פיתוח תוכנה תקין, הלקוח והספק מסכימים על מנשק נוקשה שהספק ייצא

- מכיוון שב Java המנשק הוא רכיב בשפת התכנות ועקביותו נאכפת ע"י המהדר, אנו מקבלים את היתרונות הבאים:
  - גמישות בקוד הלקוח (התלוי במנשק בלבד)
  - חופש פעולה מוגדר היטב עבור הספק למימוש המנשק

מנשקים וירושה
(הערה למי שמכיר ירושה)

- לדעתנו, **אין** לראות במנשקים כחלק ממנגנון **הירושה** של Java
- בקורסי Java רבים מוצג המנשק כמקרה פרטי של abstract class (נושא שילמד בהמשך הקורס) ואולם לדעתנו הקשר זה הוא טכני בלבד
- אנו נלמד מנשקים בהקשר של תיכון מערכת תוכנה על פי יחסי ספק-לקוח
- בהקשר זה, נעמוד על חשיבותו של המנשק בהפחתת התלות בין הרכיבים השונים במערכת

מוטיבציה: מנשק עבור Point

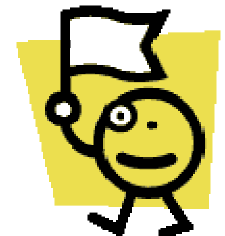
- בעת עבודה על מערכת תוכנה, הוחלט שהמערכת תכלול (בין השאר) את הרכיבים Point ו- Rectangle
- רכיבים אלו ימומשו כמחלקות בשפת Java ע"י מפתחים שונים בצוות
- לפני תחילת העבודה יש לקבוע מנשק מוסכם למחלקה Point שיקרא IPoint

המנשק IPoint

- המנשק מכיל את השרותים הציבוריים (public methods) שתספק המחלקה המבוקשת (לא בהכרח את כולם)
- המנשק אינו מכיל שרותים שאינם ציבוריים ואינו מכיל שדות (גם לא שדות public)
- המנשק אינו מכיל בנאים
- המנשק אינו מכיל מתודות static
- בשפת Java אין צורך לציין את המתודות של interface כ public אולם אנו עושים זאת לצורך בהירות

IPoint

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

public interface IPoint {

    /** returns the x coordinate of the current point*/
    public double x();

    /** returns the x coordinate of the current point*/
    public double y();

    /** returns the distance between the current point and (0,0) */
    public double rho();

    /** returns the angle between the current point and the abscissa */
    public double theta();
```



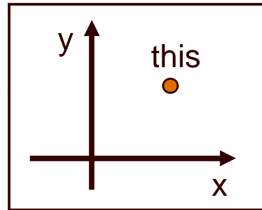
```
/** returns the distance between the current point and other */
 public double distance(IPoint other);

/** returns a point that is symmetrical to the current point
 *  with respect to X axis */
public IPoint symmetricalX();

/** returns a point that is symmetrical to the current point
 *  with respect to Y axis */
public IPoint symmetricalY();

/** returns a string representation of the current point */
public String toString();

/** move the current point by dx and dy */
public void translate(double dx, double dy);

/** rotate the current point by angle degrees with respect to (0,0) */
public void rotate(double angle);
}
```

המנשק והחוזה

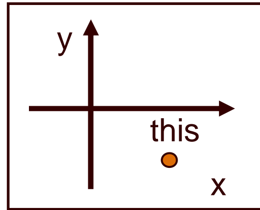
- המנשק הוא המקום האידאלי להגדרת **חוזה** ו**מצב מופשט** לטיפוס נתונים
- מכיוון שמבנה הנתונים טרם נכתב, אין חשש שפרטי מימוש "ידלפו" למפרט

- נתאר את המצב המופשט של IPoint בעזרת **תרשים**, כדי להדגים כי תיאור מופשט לא **חייב** להיות מבוטא בעזרת נוסחאות (אף על פי שבדרך כלל זו הדרך הנוחה ביותר)
- כמו כן, נציג חלוקה של **השאילתות** לשני סוגים: **צופות**, **ומפיקות**

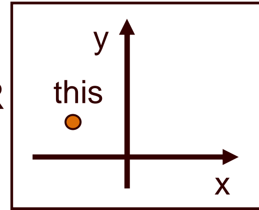
```
package il.ac.tau.cs.software1.shapes;

/** @abst
 */
```



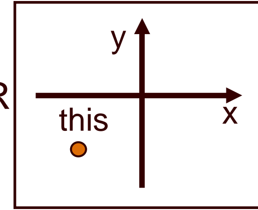
OR



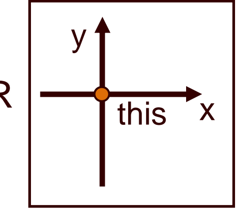
OR



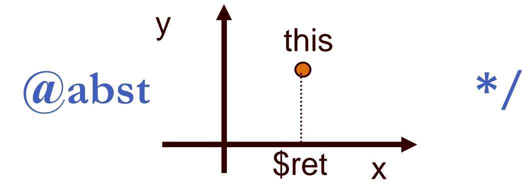
OR



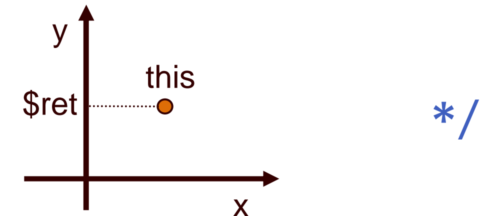
OR



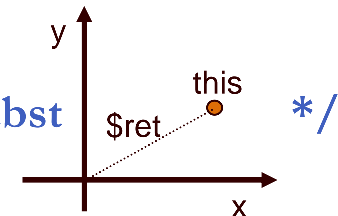

```
public interface IPoint {

    /** returns the x coordinate of the current point      @abst     */
    public double x();
```



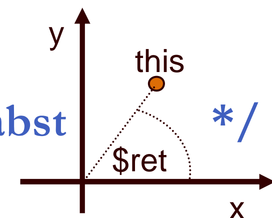

```
    /** returns the x coordinate of the current point   @abst     */
    public double y();
```




```
    /** returns the distance between the current point and (0,0)  @abst   */
    public double rho();
```




```
    /** returns the angle between the current point and the abscissa @abst   */
    public double theta();
```



תרשימים ותיעוד

- הגדרת מפרט בעזרת תרשימים מעלה מספר קשיים. למשל, היא מקשה על שילוב המפרט בגוף הקוד
- סוגיית הטכנולוגיה יכולה להיפתר בכמה דרכים. למשל:

```
/**

    ^
    |   *
    |  /
    | /
---+---+------>
    | $ret

*/
public double x();
```

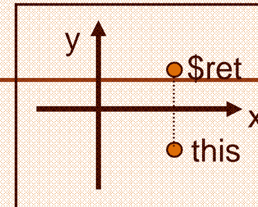
- אפשרות אחרת היא שילוב התמונות בגוף הערות ה javadoc אשר תומך ב HTML

הצופים והצופות

- השאילתות (queries) **`x(), y(), rho(), theta()`** הן **צופות** (observers)
  - הן מחזירות חיווי כלשהו על העצם שאותו הן מתארות
  - הערך המוחזר אינו מהטיפוס שעליו הן פועלות
- קיימות שאילתות אחרות המכונות **מפיקות** (producers):
  - הן מחזירות עצם מהטיפוס שעליו הן פועלות
  - הן לא משנות את העצם הנוכחי
  - לדוגמא, פעולת ה'+' לא משנה את הארגומנט שעליו היא פועלת:

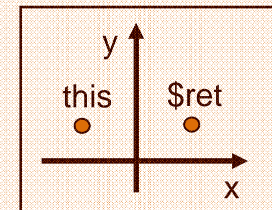
```
int x = 1
int y = x + 2;
```

```
/** returns a point that is symmetrical to the current point
 *  with respect to X axis */
public IPoint symmetricalX();
```

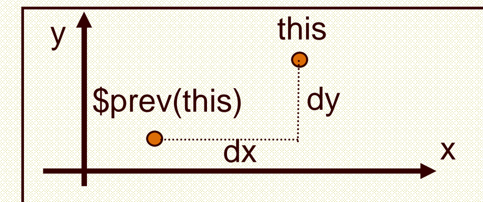



```
/** returns a point that is symmetrical to the current point
 *  with respect to Y axis */
public IPoint symmetricalY();
```

מפיקות

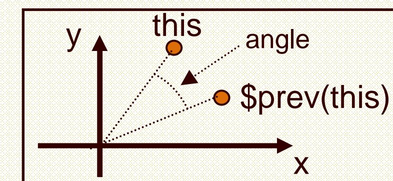



```
/** move the current point by dx and dy */
public void translate(double dx, double dy);
```




```
/** rotate the current point by angle degrees with respect
 *   to (0,0) */
public void rotate(double angle);
}
```

פקודות



לקוחות של מנשק

- בהמשך נציג 3 מימושים שונים למנשק IPoint
- ואולם כבר כעת יכול כותב המחלקה Rectangle, שהוא לקוח של המנשק IPoint, לכתוב את קוד הלקוח (כמעט) במלואו ללא תלות במימוש הספציפי
- הערה: גם כותב המחלקה Rectangle יכול היה להגדיר מנשק (אולי IRectangle) לשימוש מחלקות אחרות, ואולם כדי לפשט את הדוגמא נציג ישר את המחלקה (ונתחרט על כך אחר כך...)

האצלה

- כתיבה נכונה של שרותי המלבן תעשה שימוש בשירותי נקודה
- כל פעולה/שאילתה על מלבן "תיתרגם" לפעולות/שאילתות על קודקודיו
- הדבר יוצר את ה**הכמסה** וה**הפשטה** (encapsulation and abstraction) המאפיינות תוכנה מונחית עצמים
- הרקורסיביות הזו (רדוקציה) נקראת **האצלה** (delegation) או **פעפוע** (propagation)

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

public class Rectangle {

    private IPoint topRight;
    private IPoint bottomLeft;

    /** constructor using points */
    public Rectangle(IPoint bottomLeft, IPoint topRight) {
        this.bottomLeft = bottomLeft;
        this.topRight = topRight;
    }

    /** constructor using coordinates */
    public Rectangle(double x1, double y1, double x2, double y2) {
        topRight = ???;
        bottomLeft = ???;
    }
```

```java
/** returns a point representing the bottom-right corner of the rectangle*/
public IPoint bottomRight() {
    return ???;
}

/** returns a point representing the top-left corner of the rectangle*/
public IPoint topLeft() {
    return ???;
}

/** returns a point representing the top-right corner of the rectangle*/
public IPoint topRight() {
    return ???;
}

/** returns a point representing the bottom-left corner of the rectangle*/
public IPoint bottomLeft() {
    return ???;
}
```

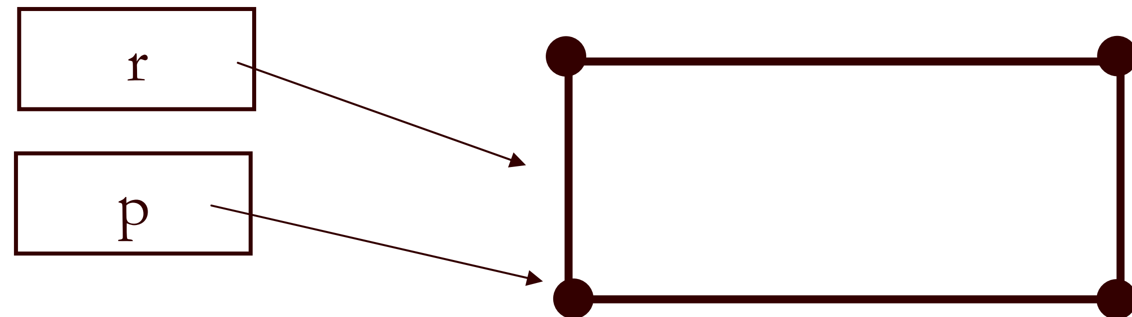
```
/** returns a point representing the bottom-right corner of the rectangle*/
public IPoint bottomRight() {
    return ???;
}

/** returns a point representing the top-left corner of the rectangle*/
public IPoint topLeft() {
    return ???;
}
```

```
/** returns a point representing the top-right corner of the rectangle*/
public IPoint topRight() {
    return topRight;
}

/** returns a point representing the bottom-left corner of the rectangle*/
public IPoint bottomLeft() {
    return bottomLeft;
}
```

מה רע באפשרות הזו?

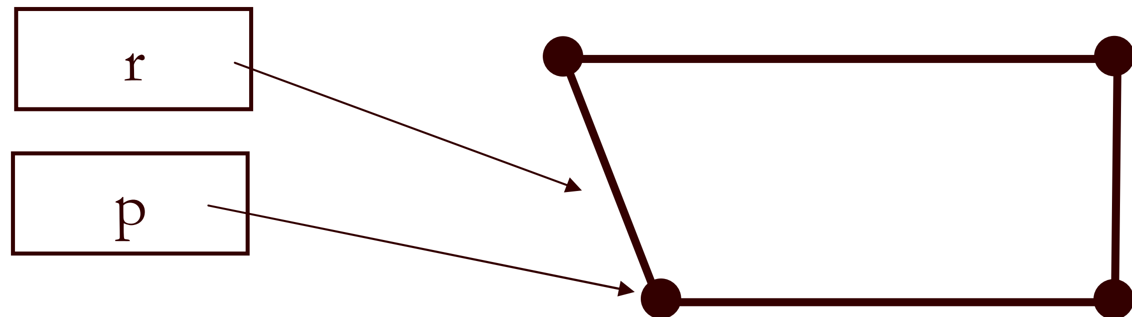
"ופרצת ופרצת..."

```
Rectangle r = new Rectangle(...);
IPoint p = r.bottomLeft();
p.translate(1.0, 0.0);
```



"ופרצת ופרצת..."

```
Rectangle r = new Rectangle(...);
IPoint p = r.bottomLeft();
p.translate(1.0, 0.0);
// here r is not a rectangle anymore...
```



משתמר המלבן

- אם היינו מנסחים בזהירות את משתמר המלבן היינו מגלים כי עבור מלבן שצלעותיו מקבילות לצירים צריך להתקיים בכל נקודת זמן:

```
/** @inv bottomLeft().x() == topLeft().x()
    @inv bottomLeft().y() == bottomRight().y()
    @inv bottomRight().x() == topRight().x()
    @inv topLeft().y() == topRight().y()
*/
public class Rectangle {
```

- החזרת נקודות הקודקוד מהשאילתות מסכנת משתמר זה
- נציג כמה דרכים להתמודד עם הבעיה