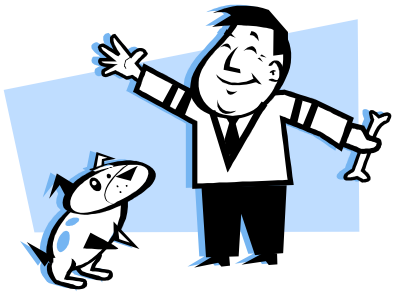
תוכנה 1 בשפת Java

שיעור מספר 4: "זרוק לו עצם"

מקוצר סמסטר ב' 2007



בית הספר למדעי המחשב
אוניברסיטת תל אביב

על סדר היום

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

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`**

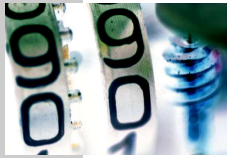
"הוכחה"

- **<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

```
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++;
   }
}
```

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");
   }
}
```

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

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

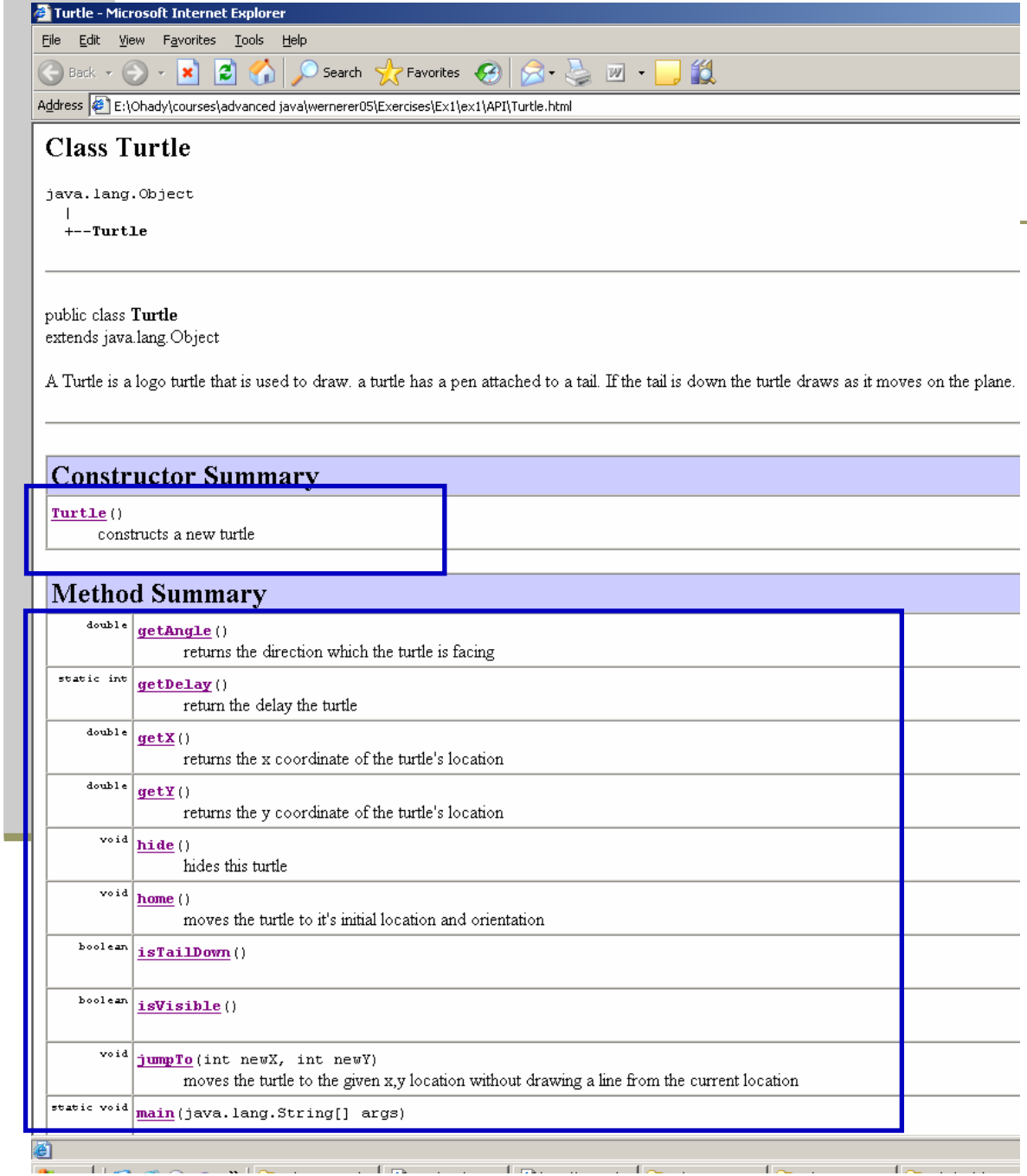
שימוש במחלקות קיימות

- 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

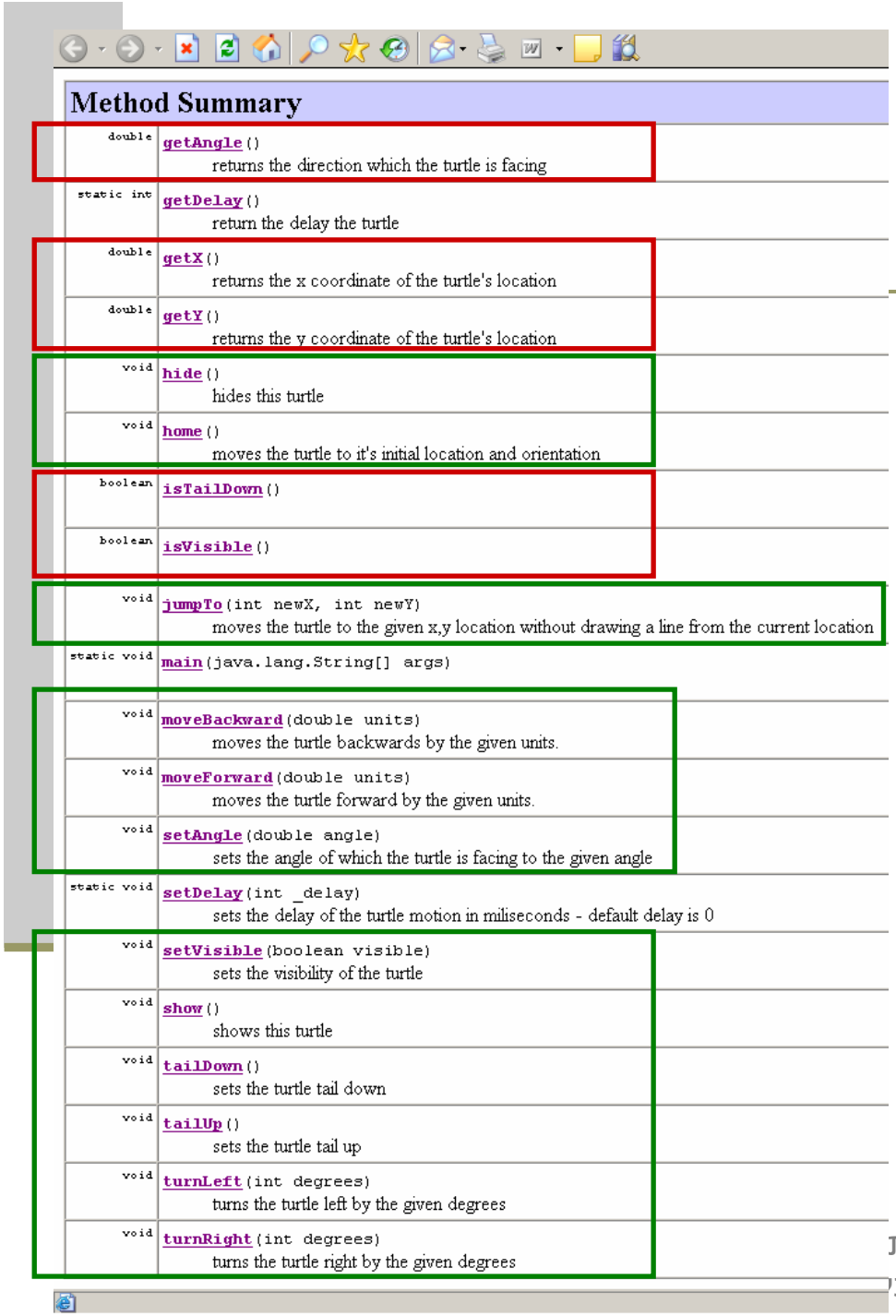


בנאי – פונקצית אתחול - ניתן לייצר מופעים חדשים של המחלקה ע"י קריאה לבנאי עם האופרטור new

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

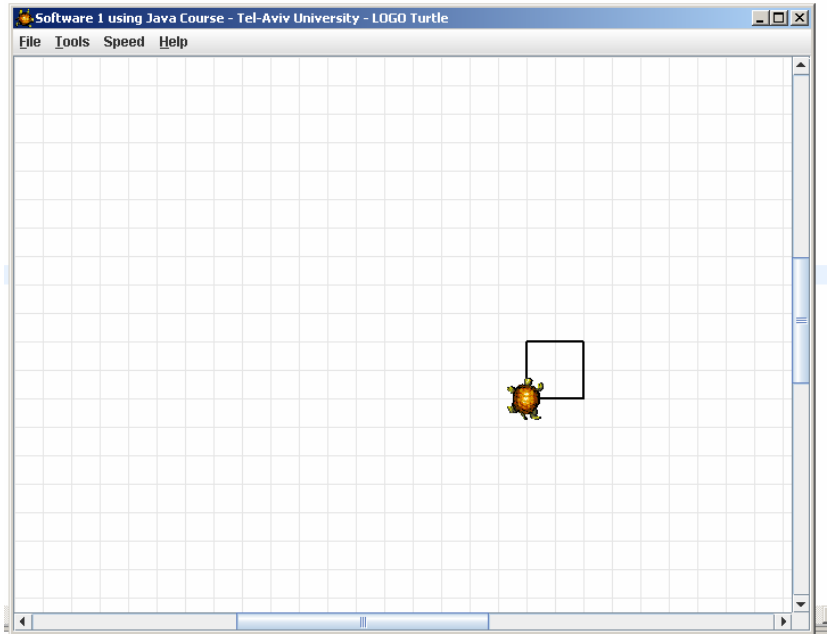
סוגים של שרותי מופע:

1. **שאילתות** (queries) –
   - שרותים שיש להם ערך מוחזר
   - בדרך כלל לא משנים את מצב העצם)
   - בשיעור הבא נדון בסוגים שונים של שאילתות

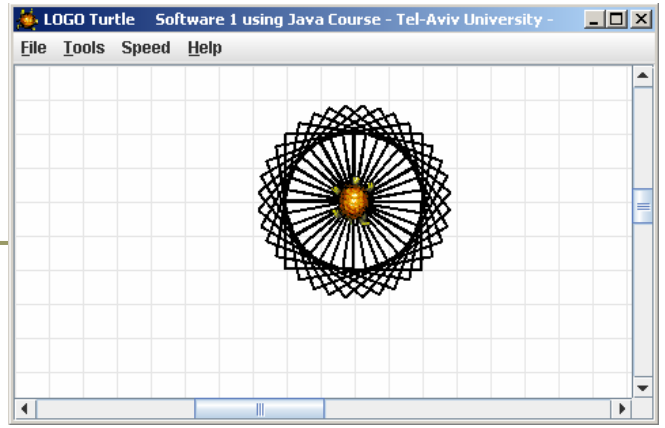
2. **פקודות** (commands) –
   - שרותים ללא ערך מוחזר
   - בדרך כלל משנים את מצב העצם שעליו הם פועלים

דוגמת שימוש

```
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**

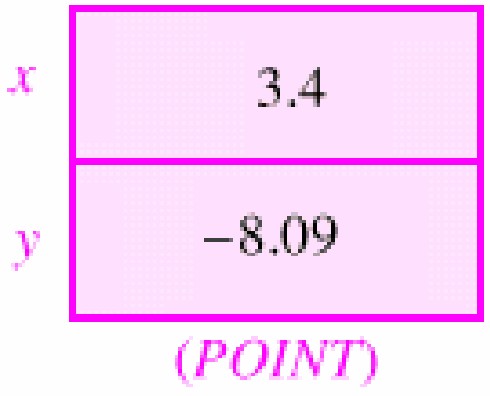
- כל אחד מהעצמים שיווצר מהמחלקה יכיל שדות מופע משלו, שערכיהם אינם תלויים בשדות המופע של עצמים אחרים

- 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...
}
```

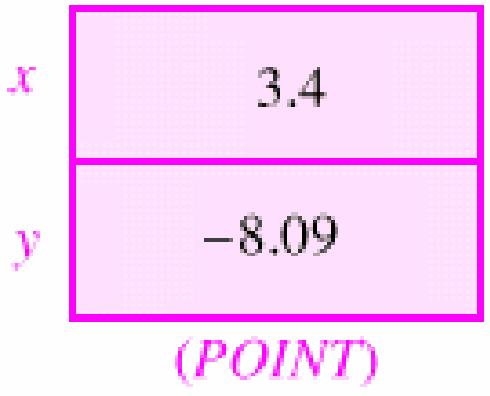
כאשר פונים לשדה מופע או שרות מופע ניתן להשתמש בהפניה this הנוצרת אוטומטית ע"י הקומפיילר של Java



POINT Class

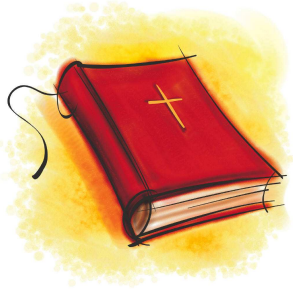
```
public class POINT {
    private double x;
    private double y;

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

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

    // More Mothods...
}
```

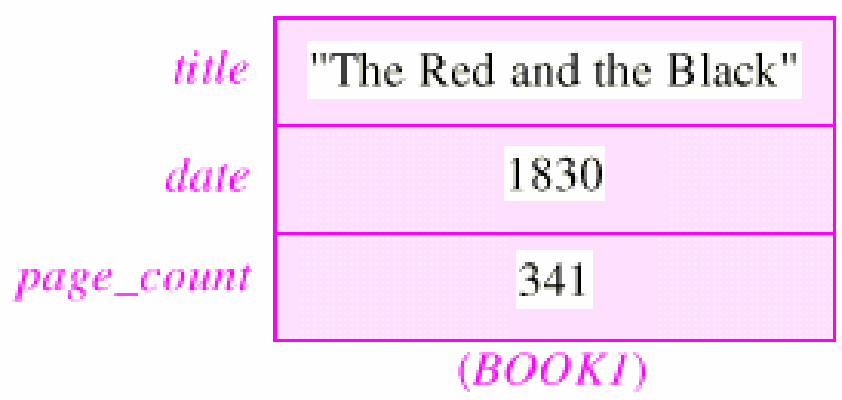
הדבר שימושי אם שם ארגומנט
לשרות זהה לשם של שדה מופע



Simple Book

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

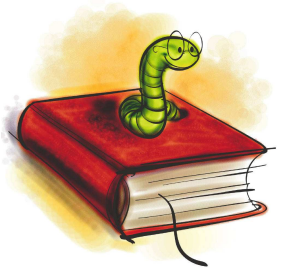


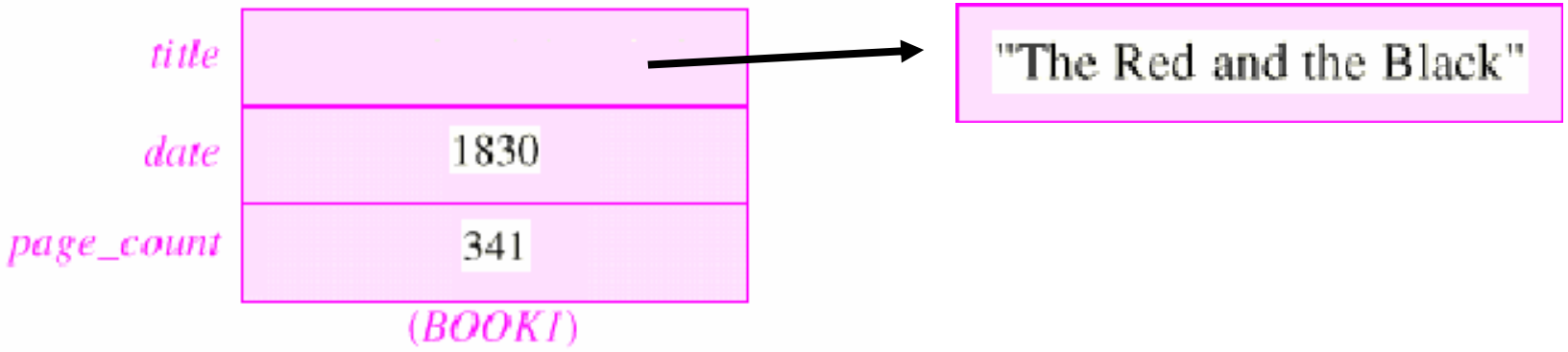


התרשים פשטני –
מחרוזת היא עצם ולכן
השדה title מכיל רק
הפנייה אליו

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*)

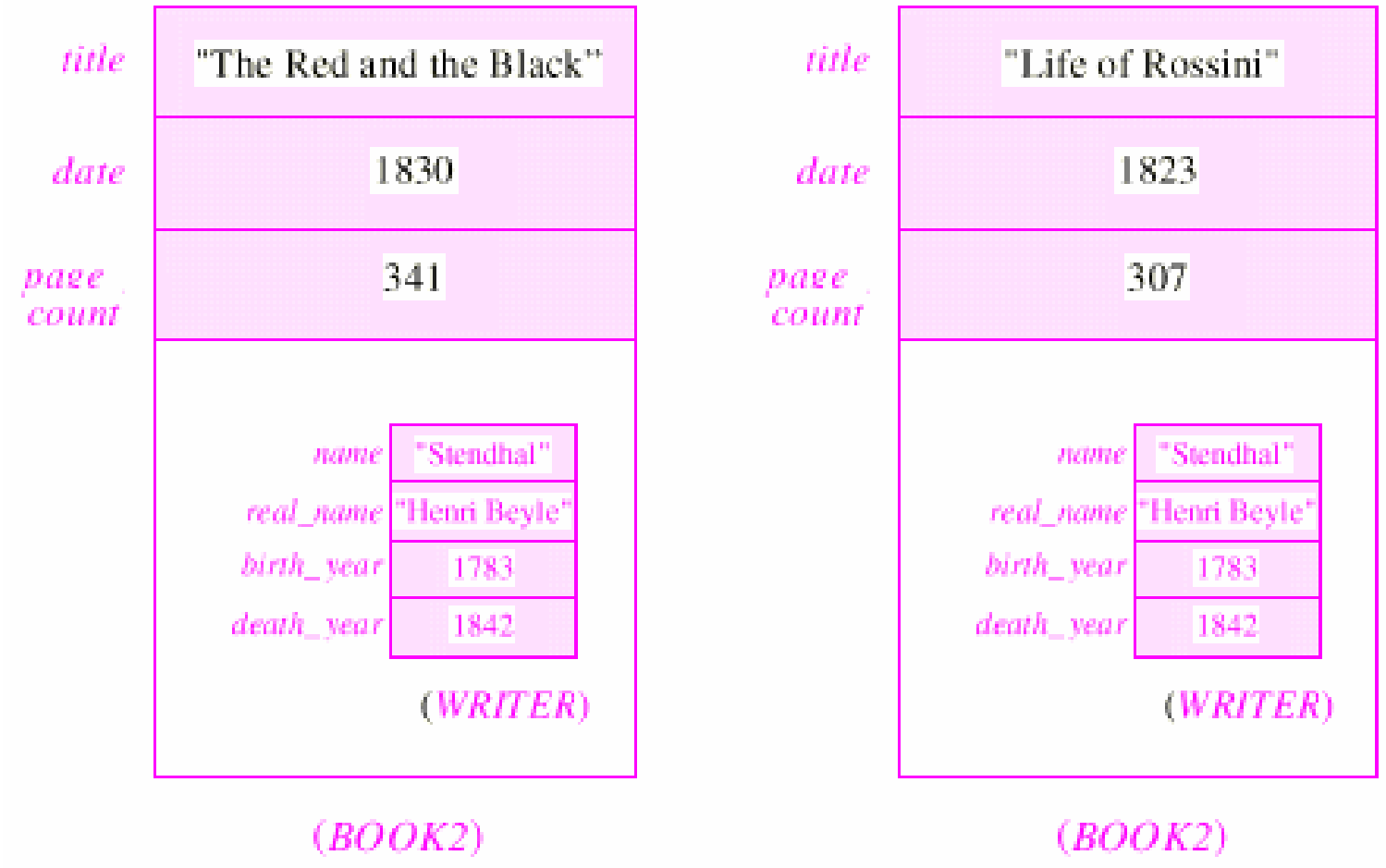
עצמים המתייחסים לעצמים

- איך נבטא את הקשר שבין ספר ומחברו?

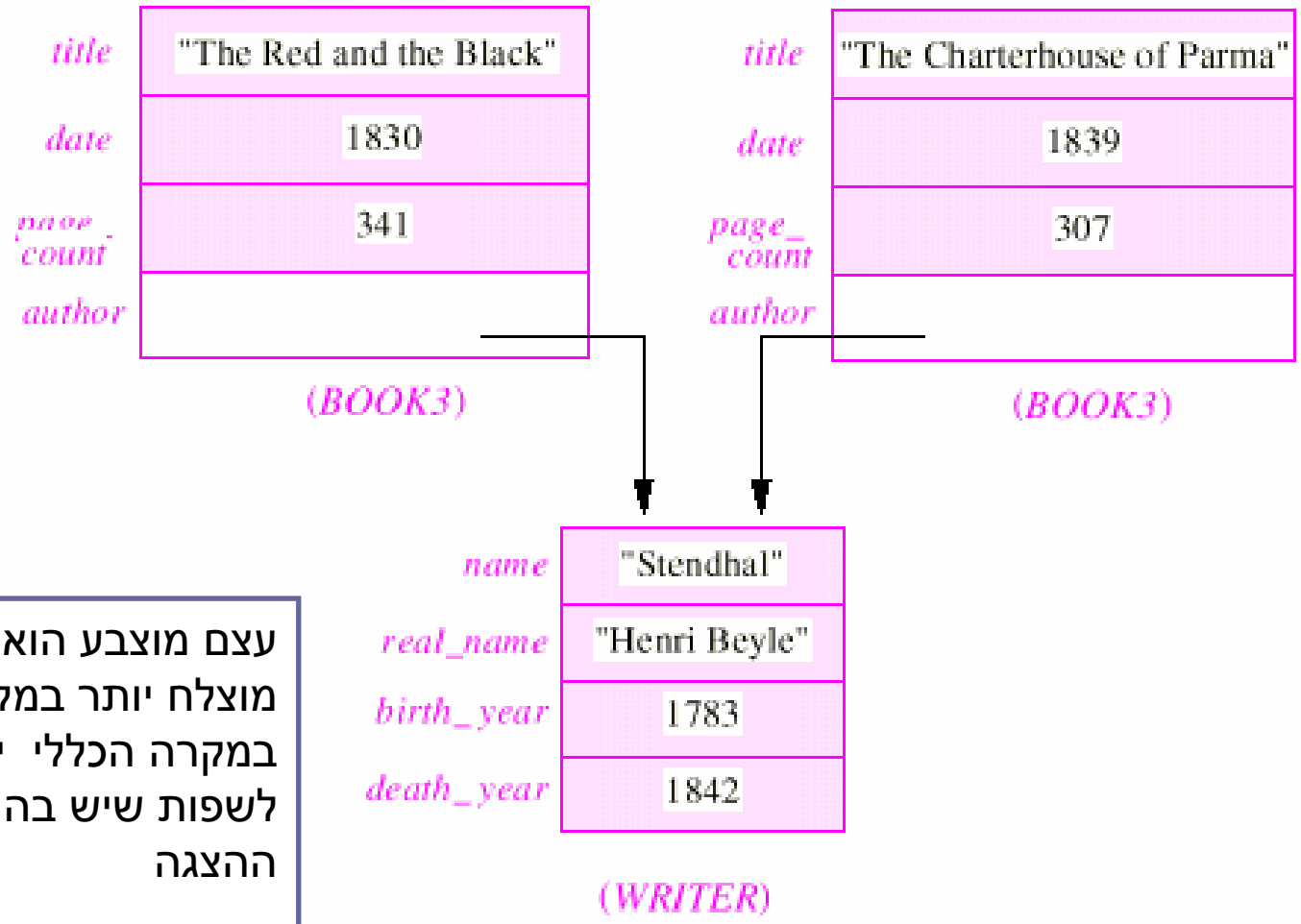
```
public class BOOK3 {
    private String title;
    private int date;
    private int page_count;
    private Writer author;
}
```

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

עצם מוכל (לא ב- Java)



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

הכלה או הצבעה?

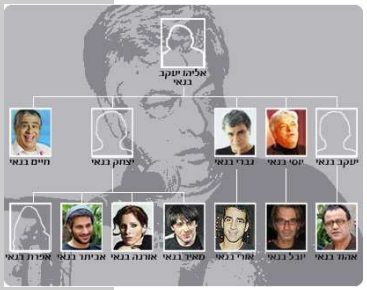
- בשפת Java הוחלט **שלא לאפשר** הכלת עצמים
- כל ההתייחסויות לעצמים בשפה הן הפניות
- הדבר מצריך משנה זהירות במקרים של **שיתוף** עצמים (sharing, aliasing)
- ניתן להתמודד עם קושי זה בעזרת אכיפה של **קיבעון** (immutability) כפי שנראה בשיעור הבא

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)