תוכנה 1 בשפת Java

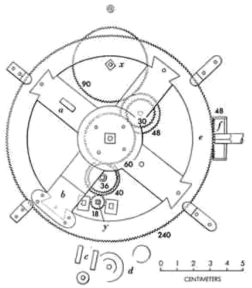
שיעור מספר 3: "מחלקות וטיפוסים"

בית הספר למדעי המחשב

אוניברסיטת תל אביב

על סדר היום

- מודל הזיכרון של Java
  - Heap and Stack
  - העברת ארגומנטים
- מנגנוני שפת Java
- עצמים ושירותי מופע



העברת ארגומנטים

- כאשר מתבצעת קריאה לשרות, ערכי הארגומנטים נקשרים לפרמטרים הפורמליים של השרות לפי הסדר, ומתבצעת השמה לפני ביצוע גוף השרות.
- בהעברת ערך לשרות הערך **מועתק** לפרמטר הפורמלי
- צורה זאת של העברת פרמטרים נקראת **call by value**
- כאשר הארגומנט המועבר הוא **הפנייה** (התייחסות, reference) העברת הפרמטר **מעתיקה את ההתייחסות.**
  - בשפות תכנות אחרות ניתן לבצע העברה של המצביע עצמו

ב Java גם reference מועבר by value

העברת פרמטרים by value

- העברת פרמטרים by value (ע"י העתקה) יוצרת מספר מקרים מבלבלים, שידרשו מאיתנו הכרות מעמיקה יותר עם מודל הזיכרון של Java
- למשל, מה מדפיס הקוד הבא?

```
public class CallByValue {

   public static void setToFive(int arg){
      arg = 5;
   }

   public static void main(String[] args) {
      int x = 0;
      System.out.println("Before: x=" + x);
      setToFive(x);
      System.out.println("After: x=" + x);
   }
}
```

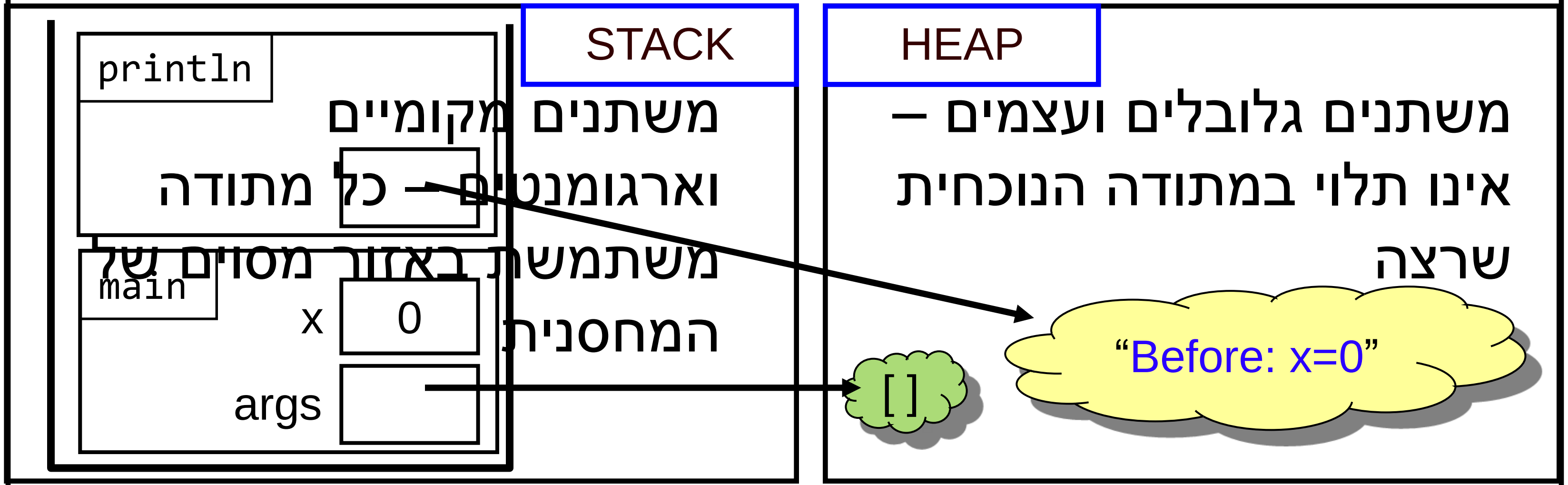
מודל הזיכרון של Java

STACK	HEAP
משתנים מקומיים וארגומנטים – כל מתודה משתמשת באזור מסוים של המחסנית	משתנים גלובלים ועצמים – אינו תלוי במתודה הנוכחית שרצה

CODE

קוד התוכנית

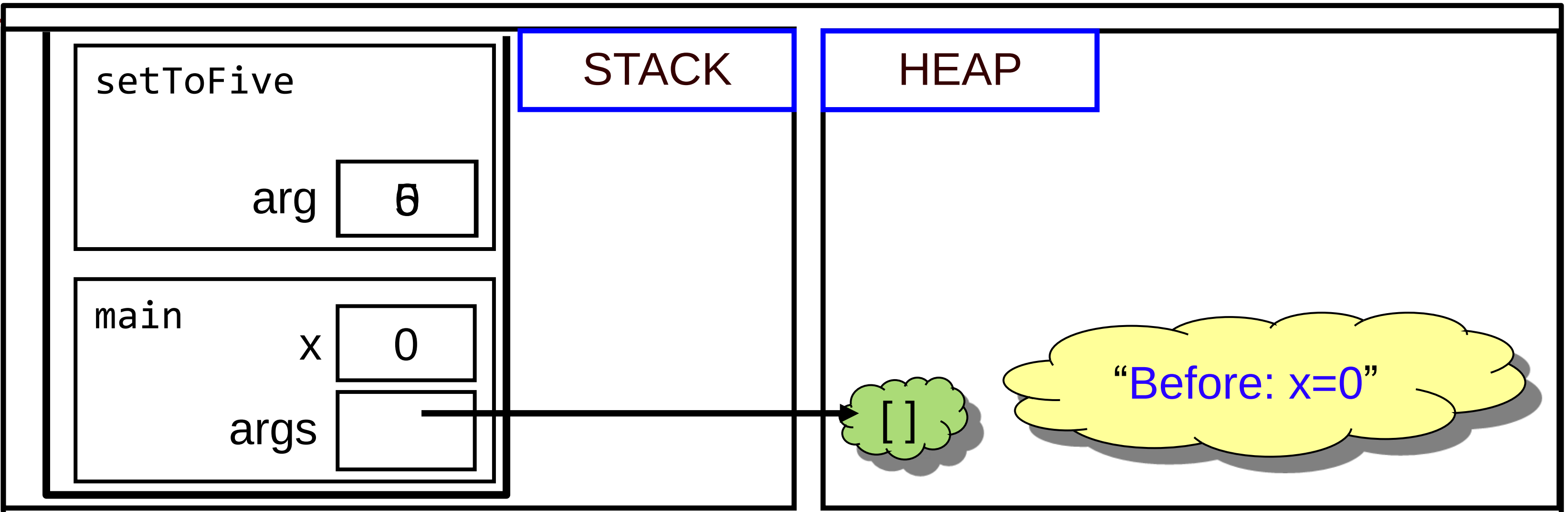
Primitives by value



CODE

```
public class CallByValue {

    public static void setToFive(int arg){
        arg = 5;
    }

    public static void main(String[] args) {
        int x = 0;
        System.out.println("Before: x=" + x);
        setToFive(x);
        System.out.println("After: x=" + x);
    }
}
```

Primitives by value

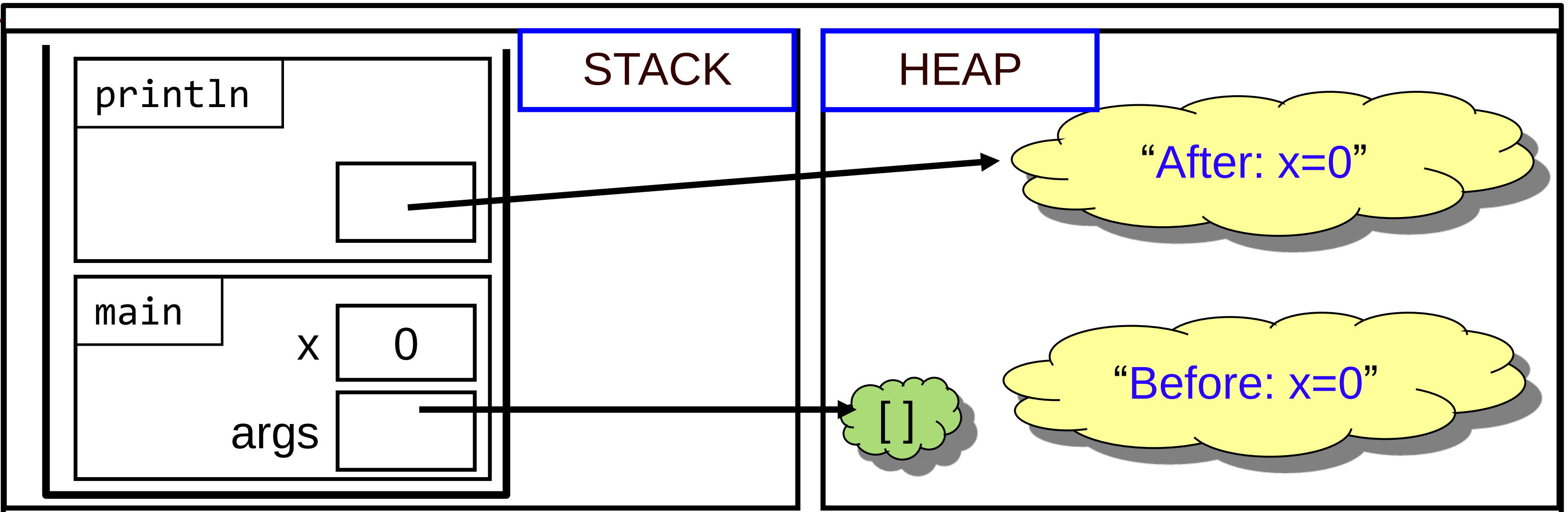


CODE

```
public class CallByValue {

    public static void setToFive(int arg){
        arg = 5;
    }

    public static void main(String[] args) {
        int x = 0;
        System.out.println("Before: x=" + x);
        setToFive(x);
        System.out.println("After: x=" + x);
    }
}
```

לאחר ש **setToFive** מסיימת את ריצתה המקום שהוקצה עבורה על ה-Stack משוחרר

Primitives by value




```java
public class CallByValue {

    public static void setToFive(int arg){
        arg = 5;
    }

    public static void main(String[] args) {
        int x = 0;
        System.out.println("Before: x=" + x);
        setToFive(x);
        System.out.println("After: x=" + x);
    }
}
```

CODE

לאחר ש **main** מסיימת את ריצתה המקום שהוקצה עבורה על ה- Stack משוחרר

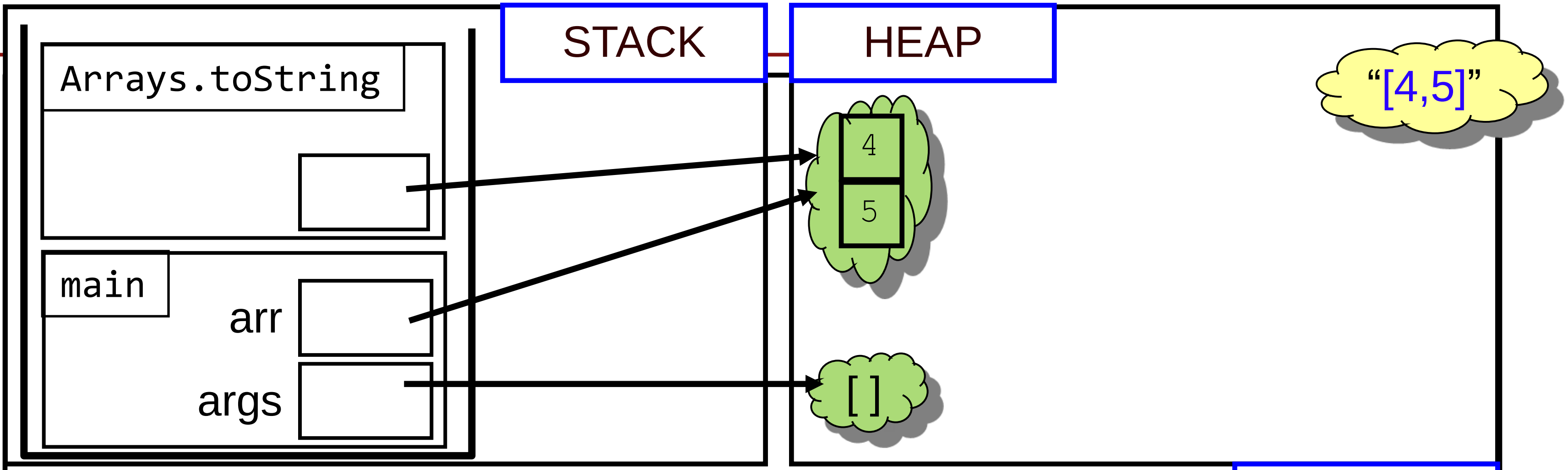
שמות מקומיים

- בדוגמא ראינו כי הפרמטר הפורמלי **arg** קיבל את הערך של הארגומנט x
- בחירת השמות השונים אינה משמעותית - יכולנו לקרוא לשני המשתנים באותו שם ולקבל התנהגות זהה
- שם של משתנה מקומי **מסתיר** משתנים בשם זהה הנמצאים בתחום עוטף או גלובלים
  - נראה את ההתנהגות הזו בהמשך השיעור, בדוגמא של *בנאים*.
- מתודה מכירה רק משתני מחסנית הנמצאים באזור שהוקצה לה על המחסנית (frame)

- מה יקרה אם המשתנה המקומי x שהועבר היה מטיפוס הפנייה? למשל, מה מדפיס הקוד הבא?

```
import java.util.Arrays; //explained later...

public class CallByValue {

    public static void setToZero(int [] arr){
        arr = new int[3];
    }

    public static void main(String[] args) {
        int [] arr = {4,5};
        System.out.println("Before: arr=" + Arrays.toString(arr));
        setToZero(arr);
        System.out.println("After: arr=" + Arrays.toString(arr));
    }
}
```

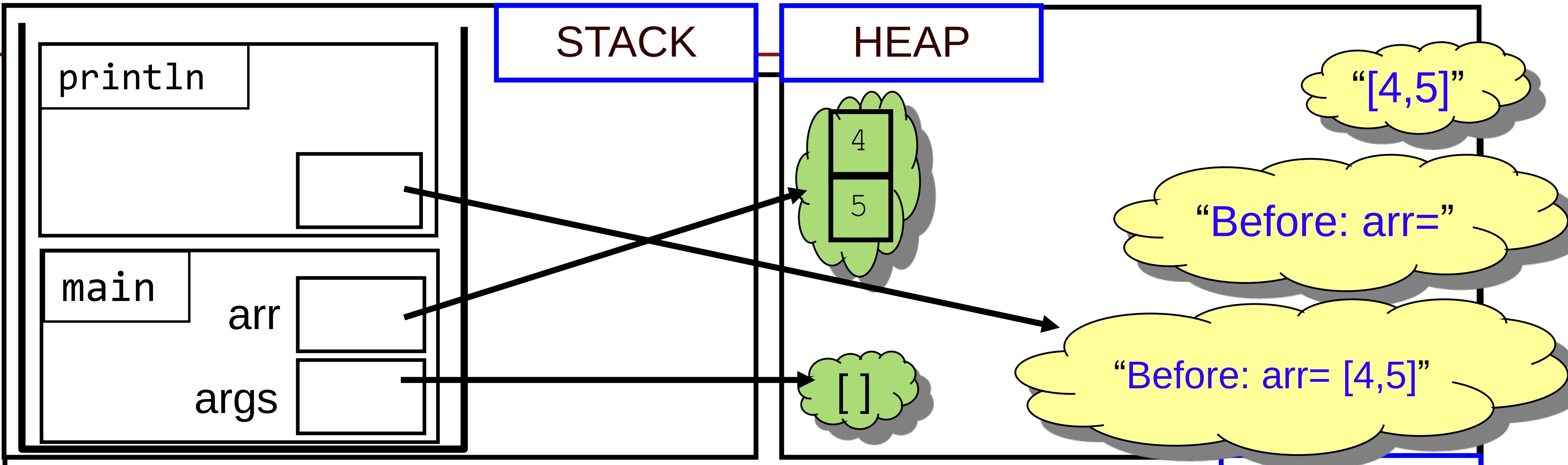
Reference by value



CODE

```
public class CallByValue {

    public static void setToZero(int [] arr){
        arr = new int[3];
    }

    public static void main(String[] args) {
        int [] arr = {4,5};
        System.out.println("Before: arr=" + Arrays.toString(arr));
        setToZero(arr);
        System.out.println("After: arr=" + Arrays.toString(arr));
    }
}
```

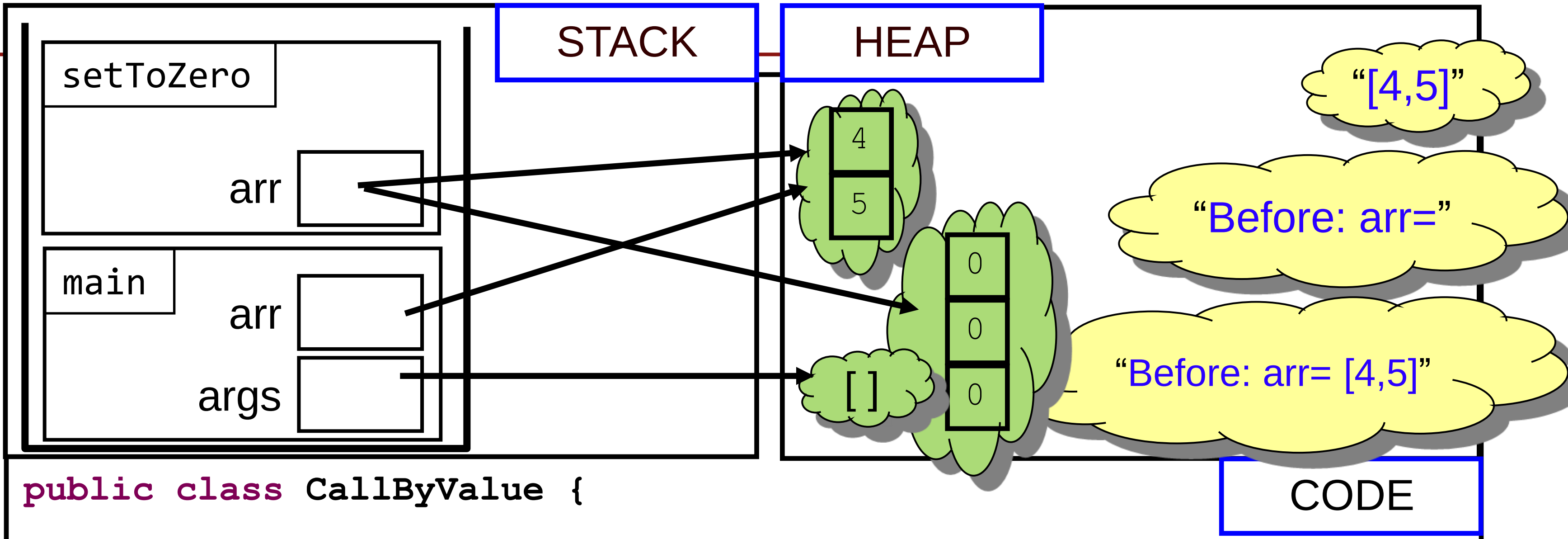
Reference by value



CODE

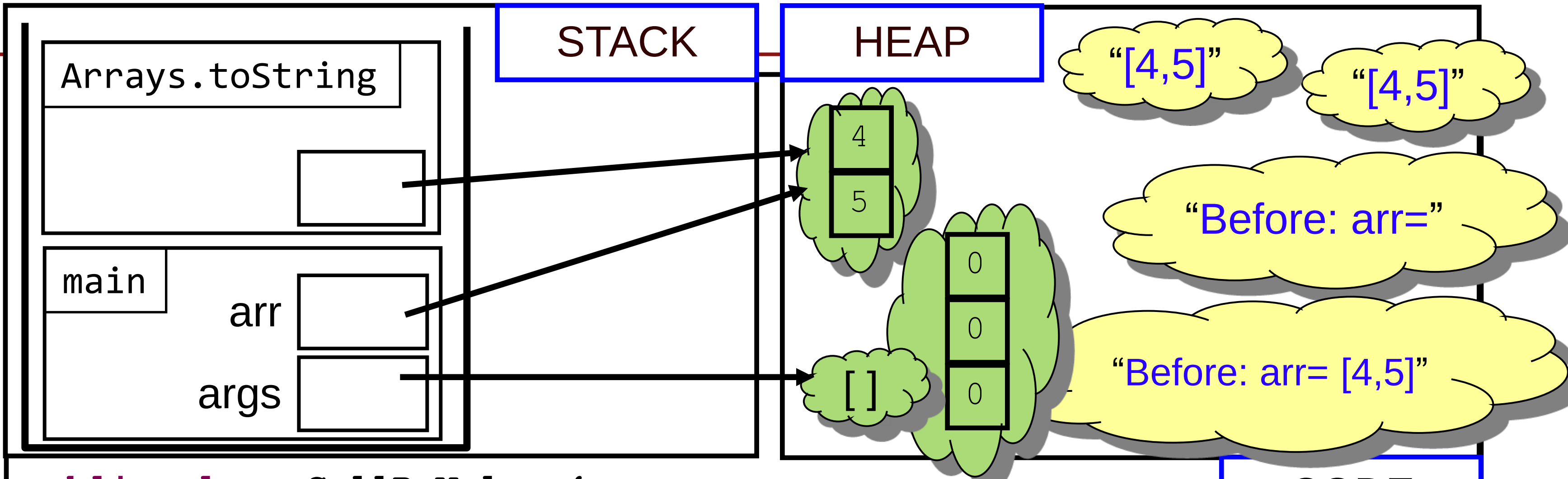
```java
public class CallByValue {

    public static void setToZero(int [] arr){
        arr = new int[3];
    }

    public static void main(String[] args) {
        int [] arr = {4,5};
        System.out.println("Before: arr=" + Arrays.toString(arr));
        setToZero(arr);
        System.out.println("After: arr=" + Arrays.toString(arr));
    }
}
```

Reference by value




```
public class CallByValue {

    public static void setToZero(int [] arr){
        arr = new int[3];
    }

    public static void main(String[] args) {
        int [] arr = {4,5};
        System.out.println("Before: arr=" + Arrays.toString(arr));
        setToZero(arr);
        System.out.println("After: arr=" + Arrays.toString(arr));
    }
}
```

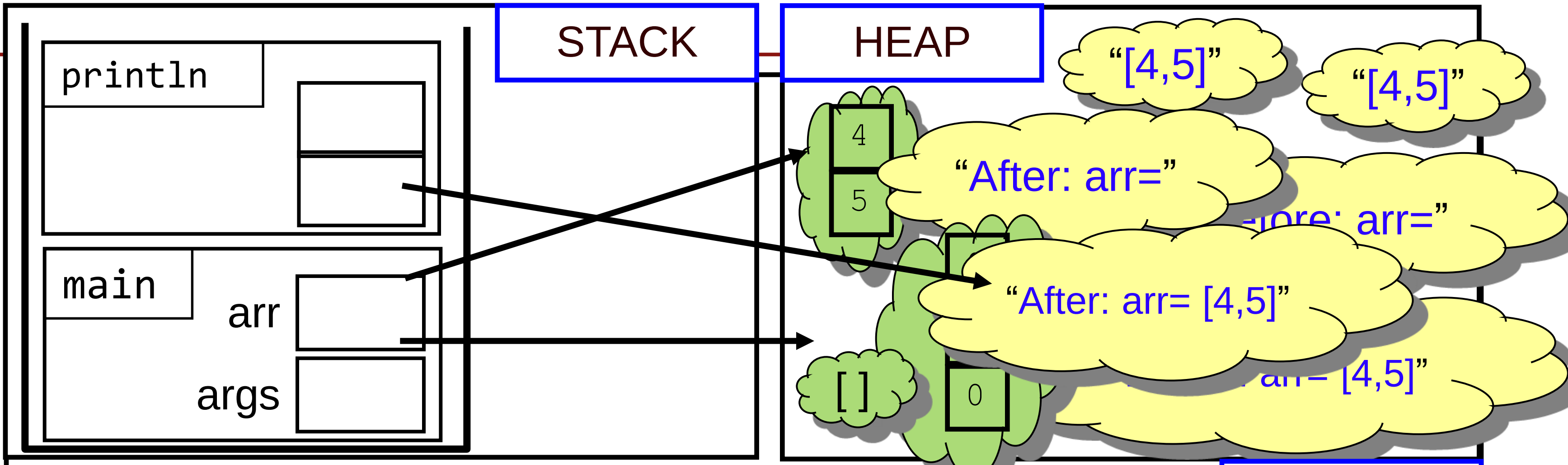
Reference by value



CODE

```
public class CallByValue {

    public static void setToZero(int [] arr){
        arr = new int[3];
    }

    public static void main(String[] args) {
        int [] arr = {4,5};
        System.out.println("Before: arr=" + Arrays.toString(arr));
        setToZero(arr);
        System.out.println("After: arr=" + Arrays.toString(arr));
    }
}
```

Reference by value



CODE

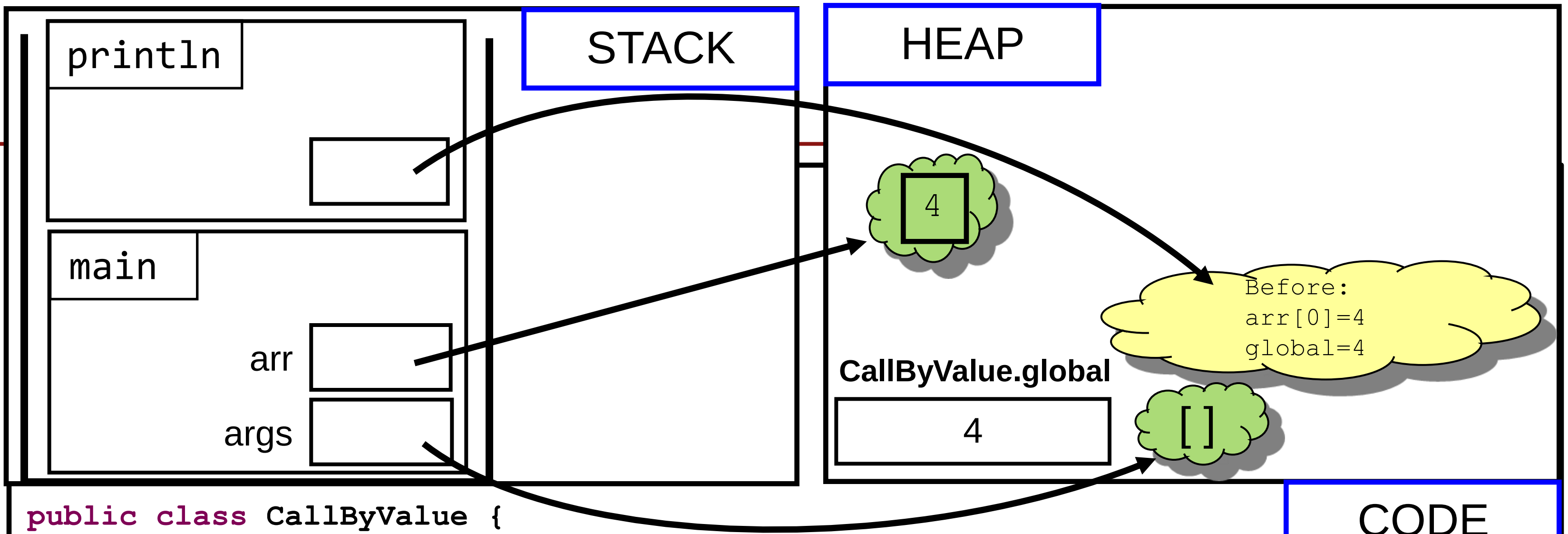
```
public class CallByValue {

    public static void setToZero(int [] arr){
        arr = new int[3];
    }

    public static void main(String[] args) {
        int [] arr = {4,5};
        System.out.println("Before: arr=" + Arrays.toString(arr));
        setToZero(arr);
        System.out.println("After: arr=" + Arrays.toString(arr));
    }
}
```

הפונקציה הנקראת והעולם שבחוץ

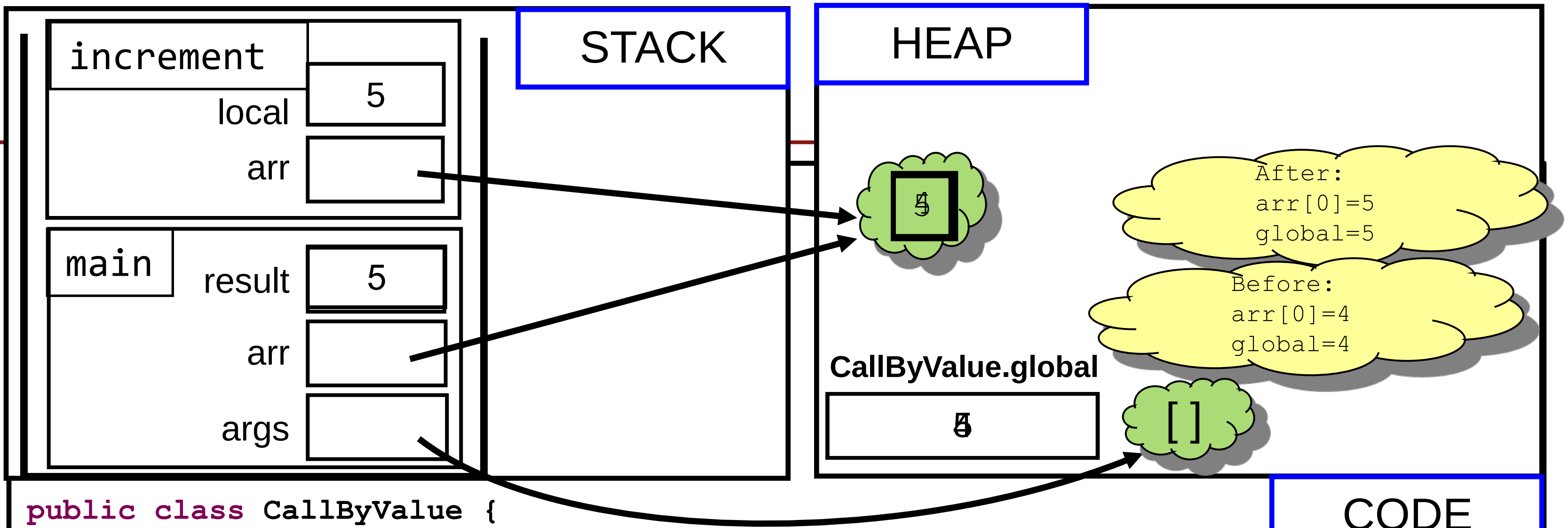
- בשיטת העברה by value לא יעזור למתודה לשנות את הארגומנט שקיבלה, מכיוון שהיא מקבלת עותק
- אז איך יכולה מתודה להשפיע על ערכים במתודה שקראה לה?
  - ע"י ערך מוחזר
  - ע"י גישה למשתנים או עצמים שהוקצו ב- Heap
- מתודות שמשנות את תמונת הזיכרון נקראות בהקשרים מסוימים Mutators או Transformers

מה מדפיסה התוכנית הבאה?

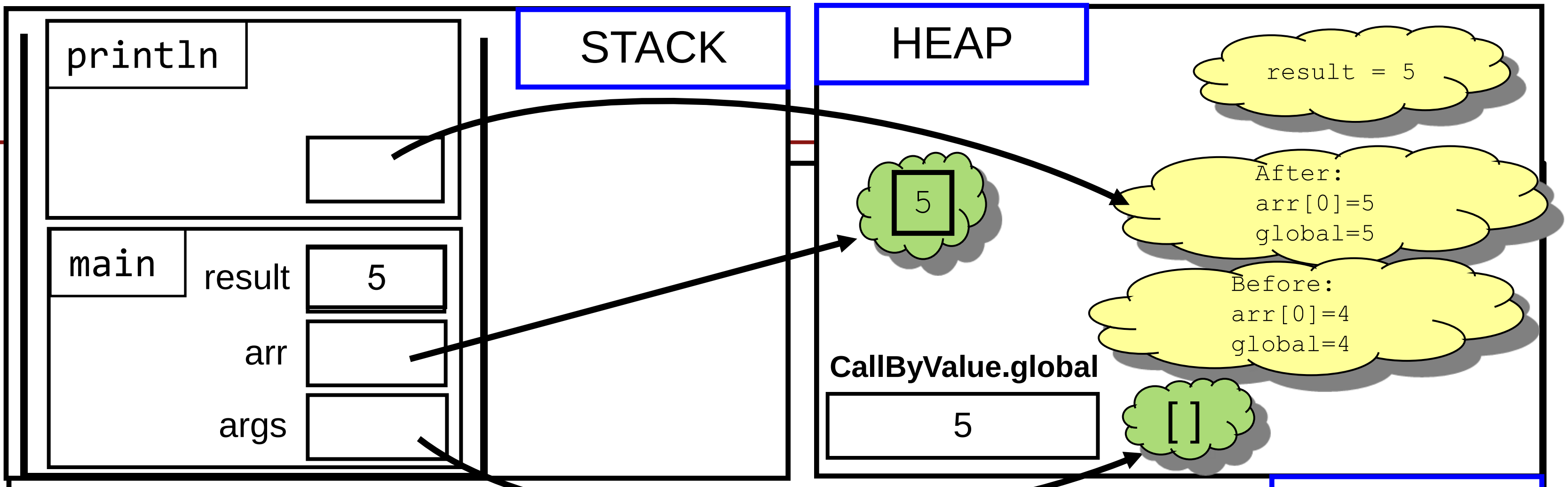
```
public class CallByValue {

    static int global = 4;

    public static int increment(int [] arr){
        int local = 5;
        arr[0]++;
        global++;
        return local;
    }

    public static void main(String[] args) {
        int [] arr = {4};
        System.out.println("Before:\narr[0]=" + arr[0] +
                                    "\nglobal=" + global);
        int result = increment(arr);
        System.out.println("After:\narr[0]=" + arr[0] +
                                    "\nglobal=" + global);
        System.out.println("result = " + result);
    }
}
```




```java
public class CallByValue {

    static int global = 4;

    public static int increment(int [] arr){
        int local = 5;
        arr[0]++;
        global++;
        return local;
    }

    public static void main(String[] args) {
        int [] arr = {4};
        System.out.println("Before:\narr[0]=" + arr[0] + "\nglobal=" + global);
        int result = increment(arr);
        System.out.println("After:\narr[0]=" + arr[0] + "\nglobal=" + global);
        System.out.println("result = " + result);
    }
}
```



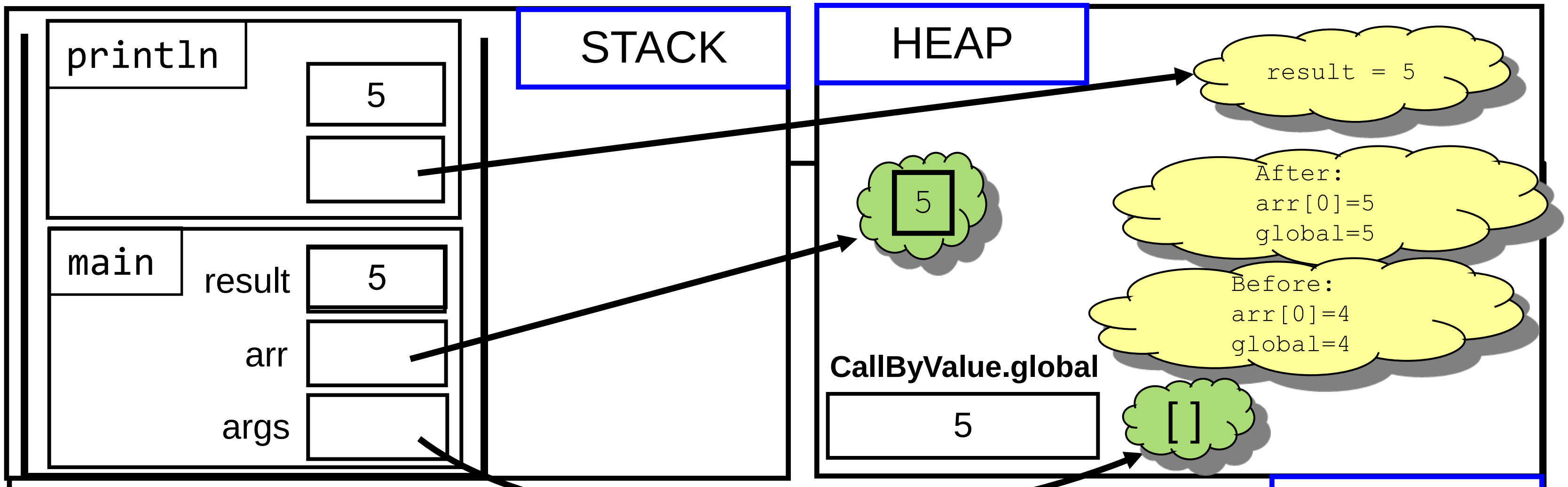

```
public class CallByValue {

    static int global = 4;

    public static int increment(int [] arr){
        int local = 5;
        arr[0]++;
        global++;
        return local;
    }

    public static void main(String[] args) {
        int [] arr = {4};
        System.out.println("Before:\narr[0]=" + arr[0] + "\nglobal=" + global);
        int result = increment(arr);
        System.out.println("After:\narr[0]=" + arr[0] + "\nglobal=" + global);
        System.out.println("result = " + result);
    }
}
```



CODE

```
public class CallByValue {

    static int global = 4;

    public static int increment(int [] arr){
        int local = 5;
        arr[0]++;
        global++;
        return local;
    }

    public static void main(String[] args) {
        int [] arr = {4};
        System.out.println("Before:\narr[0]=" + arr[0] + "\nglobal=" + global);
        int result = increment(arr);
        System.out.println("After:\narr[0]=" + arr[0] + "\nglobal=" + global);
        System.out.println("result = " + result);
    }
}
```

Heap, Heap – Hooray!



CODE

```
public class CallByValue {

     static int global = 4;

     public static int increment(int [] arr){
          int local = 5;
          arr[0]++;
          global++;
          return local;
     }

     public static void main(String[] args) {
          int [] arr = {4};
          System.out.println("Before: arr[0]=" + arr[0] + "\tglobal=" + global);
          int result = increment(arr);
          System.out.println("After:  arr[0]=" + arr[0] + "\tglobal=" + global);
          System.out.println("result = " + result);
     }
}
```

משתני פלט (Output Parameters)

- איך נכתוב פונקציה שצריכה להחזיר יותר מערך אחד?
  - הפונקציה תחזיר מערך

- ומה אם הפונקציה צריכה להחזיר נתונים מטיפוסים שונים?
  - הפונקציה תקבל כארגומנטים הפניות לעצמים שהוקצו ע"י הקורא לפונקציה (למשל הפניות למערכים), ותמלא אותם בערכים משמעותיים

- ומה קורה אם נרצה שהפונקציה לא תחזיר ערך במקרים מסויימים?
  - החל מ Java 8 – המחלקה Optional עוזרת לדמות את ההתנהגות הרצויה, נראה אותה בהמשך הקורס.

גושי אתחול סטטיים

- ראינו כי אתחול המשתנה הסטטי התרחש מיד לאחר טעינת המחלקה לזיכרון, עוד לפני פונקציית ה main
- ניתן לבצע פעולות נוספות (בדרך כלל אתחולים למניהם) מיד לאחר טעינת המחלקה לזיכרון, פעולות אלו יש לציין בתוך בלוק **`static`**
  - פרטים נוספים:

https://docs.oracle.com/javase/tutorial/java/javaOO/initial.html

תמונת הזיכרון האמיתית

- מודל הזיכרון שתואר כאן הוא פשטני – פרטים רבים נוספים נשמרים על המחסנית וב- Heap
- תמונת הזיכרון האמיתית והמדויקת היא תלוית סביבה ועשויה להשתנות בריצות בסביבות השונות
- נושא זה נידון בהרחבה בקורס "קומפילציה"

המחלקה כספריה של שרותים

המחלקה כספריה של שרותים

- ניתן לראות במחלקה **ספריה של שרותים, מודול**: אוסף של פונקציות עם מכנה משותף

- רוב המחלקות ב Java, נוסף על היותן **ספריה**, משמשות גם **כטיפוס נתונים**. ככאלו הן מכילות רכיבים נוספים פרט לשרותי מחלקה. נדון במחלקות אלו בהמשך השיעור

- ואולם קיימות ב- Java גם כמה מחלקות המשמשות כספריות בלבד. בין השימושיות שבהן:
  - **`java.lang.Math`**
  - **`java.util.Arrays`**
  - **`java.lang.System`**

חבילות ומרחב השמות

- מרחב השמות של Java הוא היררכי
  - בדומה לשמות תיקיות במערכת הקבצים

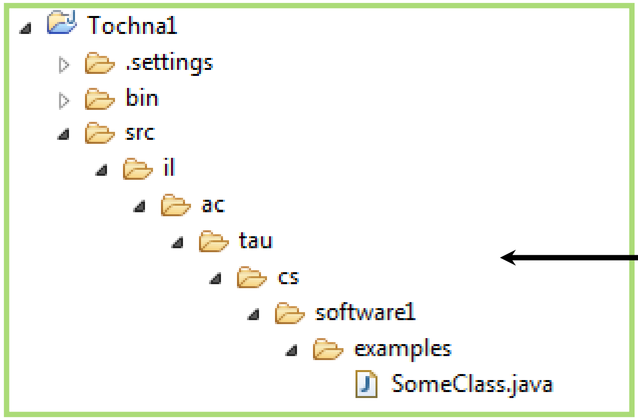
- חבילה (package) יכולה להכיל מחלקות או תת-חבילות בצורה רקורסיבית

- שמה המלא של מחלקה (fully qualified name) כולל את שמות כל החבילות שהיא נמצאת בהן מהחיצונית ביותר עד לפנימית. שמות החבילות מופרדים בנקודות

- מקובל כי תוכנה הנכתבת בארגון מסוים משתמש בשם התחום האינטרנטי של אותו ארגון כשם החבילות העוטפות

חבילות ומרחב השמות

- קיימת התאמה בין מבנה התיקיות (directories, folders) בפרויקט תוכנה ובין חבילות הקוד (packages)

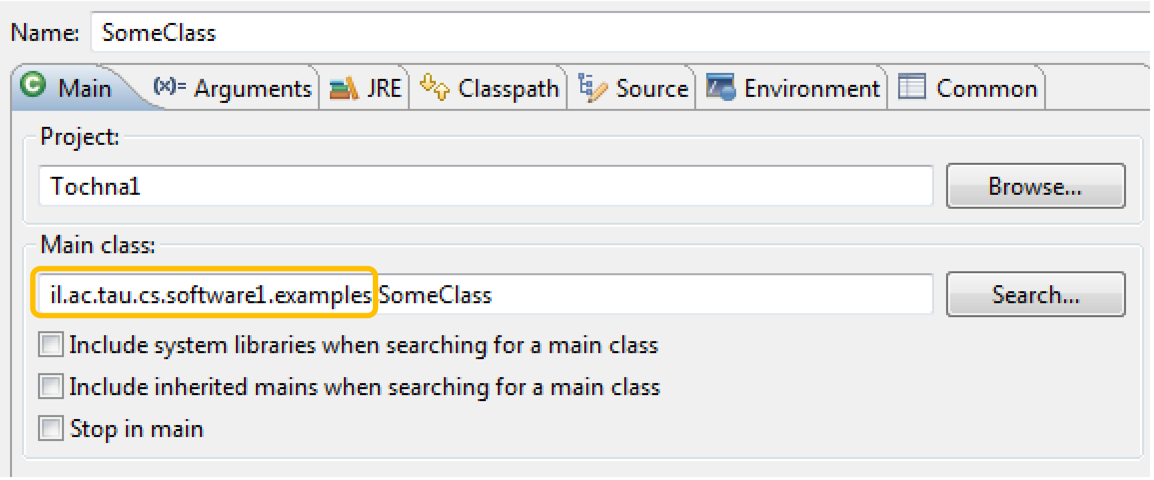


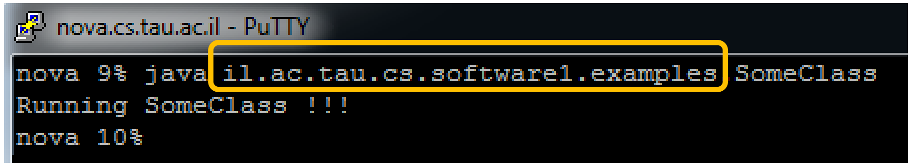

```java
package il.ac.tau.cs.software1.examples;

public class SomeClass{
    public static void main(String[] args){
        System.out.println("Running SomeClass !!!");
    }
}
```

חבילות ומרחב השמות

- כיצד נריץ תוכנית אשר נמצאת ב package כלשהו?

ב eclipse





ב command line

משפט import

- שימוש בשמה המלא של מחלקה מסרבל את הקוד:

```
System.out.println("Before: x=" +
    java.util.Arrays.toString(arr));
```

- ניתן לחסוך שימוש בשם מלא ע"י ייבוא השם בראש הקובץ (מעל הגדרת המחלקה)

```
import java.util.Arrays;
...
    System.out.println("Before: x=" + Arrays.toString(arr));
```

משפט import

- כאשר עושים שימוש נרחב במחלקות מחבילה מסויימת ניתן לייבא את שמות כל המחלקות במשפט import יחיד:

```
import java.util.*;
...
    System.out.println("Before: x=" + Arrays.toString(arr));
```

- השימוש ב-* אינו רקורסיבי, כלומר יש צורך במשפט **import** נפרד עבור כל תת חבילה:

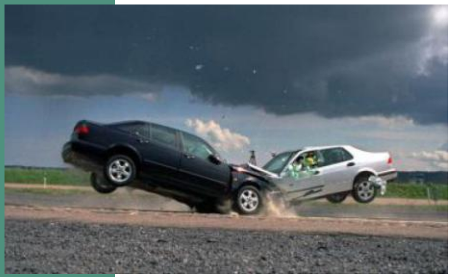
```
// for classes directly under subpackage
import package.subpackage.*;

// for classes directly under subsubpackage1
import package.subpackage.subsubpackage1.*;

// only for the class SomeClass
import package.subpackage.subsubpackage2.SomeClass;
```

משפט static import

- החל מ Java5 ניתן לייבא למרחב השמות את השרות או המשתנה הסטטי (static import) ובכך להימנע מציון שם המחלקה בגוף הקוד:

```
package il.ac.tau.cs.software1.examples;
import static il.ac.tau.cs.software1.examples.SomeOtherClass.someMethod;

public class SomeClass {

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

- גם ב static import ניתן להשתמש ב- *



הערות על מרחב השמות ב- Java

- שימוש במשפט import **אינו** שותל קוד במחלקה והוא נועד לצורכי נוחות בלבד
- אין צורך לייבא מחלקות מאותה חבילה
- אין צורך לייבא את החבילה **`java.lang`**
- ייבוא כוללני מדי של שמות מעיד על צימוד חזק בין מודולים
- ייבוא של חבילות עם מחלקות באותו שם יוצר ambiguity של הקומפיילר וגורר טעות קומפילציה ("התנגשות שמות")
- סביבות הפיתוח המודרניות יודעות לארגן בצורה אוטומטית את משפטי ה import כדי להימנע מייבוא גורף מדי ("name pollution")

CLASSPATH

- איפה נמצאות המחלקות?
- איך יודעים הקומפיילר וה- JVM היכן לחפש את המחלקות המופיעות בקוד המקור או ה byte code?

- קיים משתנה סביבה בשם **CLASSPATH** המכיל שמות של תיקיות במערכת הקבצים שם יש לחפש מחלקות הנזכרות בתוכנית
- ה- **CLASSPATH** מכיל את תיקיות ה"שורש" של חבילות המחלקות
- ניתן להגדיר את המשתנה בכמה דרכים:
  - הגדרת המשתנה בסביבה (תלוי במערכת ההפעלה)
  - הגדרה אד-הוק – ע"י הוספת תיקיות חיפוש בשורת הפקודה (בעזרת הדגל cp או classpath)
  - הגדרת תיקיות החיפוש בסביבת הפיתוח

jar

- כאשר ספקי תוכנה נותנים ללקוחותיהם מספר גדול של מחלקות הם יכולים לארוז אותן כארכיב

- התוכנית **jar** (Java **AR**chive) אורזת מספר מחלקות לקובץ אחד תוך שמירה על מבנה החבילות הפנימי שלהן

- הפורמט תואם למקובל בתוכנות דומות כגון zip, tar, rar ואחרות

- כדי להשתמש במחלקות הארוזות אין צורך לפרוס את קובץ ה- **jar**
  - ניתן להוסיפו ל **CLASSPATH** של התוכנית

- התוכנית **jar** היא חלק מה- JDK וניתן להשתמש בה משורת הפקודה או מתוך סביבת הפיתוח

API and Javadoc

- קובץ ה- jar עשוי שלא להכיל קובצי מקור כלל, אלא רק קובצי class (למשל משיקולי זכויות יוצרים)
- איך יכיר לקוח שקיבל jar מספק תוכנה כלשהו את הפונקציות והמשתנים הנמצאים בתוך ה- jar, כדי שיוכל לעבוד איתם?
- בעולם התוכנה מקובל לספק ביחד עם הספריות גם מסמך תיעוד, המפרט את שמות וחתימות המחלקות, השרותים והמשתנים יחד עם תיאור מילולי של אופן השימוש בהם
- תוכנה בשם **`javadoc`** מחוללת **תיעוד אוטומטי** בפורמט html על בסיס הערות התיעוד שהופיעו בגוף קובצי המקור
- תיעוד זה מכונה API (**A**pplication **P**rogramming **I**nterface)
- תוכנת ה **`javadoc`** היא חלק מה- JDK וניתן להשתמש בה משורת הפקודה או מתוך סביבת הפיתוח

Javadoc

```java
/** Documentation for the package */
package somePackage;

/** Documentation for the class
 * @author your name here
 */
public class SomeClass {

    /** Documentation for the class variable */
    public static int someVariable;

    /** Documentation for the class method
     * @param x documentation for parameter x
     * @param y documentation for parameter y
     * @return
     *     documentation for return value
     */
    public static int someMethod(int x, int y, int z){
        // this comment would NOT be included in the documentation
        return 0;
    }
}
```

Javadoc

כך זה ניראה ב eclipse:

```java
public class SomeClass {

    /** Documentation for the class variable */
    public static int someVariable;

    /**
     * Documentation for the class method
     *
     * @param x
     *            documentation for parameter x
     * @param y
     *            documentation for parameter y
     * @return documentation for return value
     */
    public static int someMethod(int x, int y, int z) {
        // this commen
        return 0;
    }

    public static int
        return 0;
    }
}
```

int somePackage.SomeClass.someMethod(int x, int y, int z)

Documentation for the class method

Parameters:
x documentation for parameter x
y documentation for parameter y
z
Returns:
documentation for return value

Press 'F2' for focus

Method Detail

someMethod

```
public static int someMethod(int x,
                             int y,
                             int z)
```

Documetntaion for the class method

Parameters:

`x` - `documentation for parameter x`

`y` - `documentation for parameter y`

Returns:

`documentation for return value`

חלק מדף ה html שנוצר

Java API

- ניתן למצוא את התיעוד של כל ספריות ה Java באמצעות javadoc באתר של חברת Oracle.

https://docs.oracle.com/javase/8/docs/api/

תיעוד וקוד

- בעזרת מחולל קוד אוטומטי הופך התיעוד לחלק בלתי נפרד מקוד התוכנית
- הדבר משפר את הסיכוי ששינויים עתידיים בקוד יופיעו מיידית גם בתיעוד וכך תשמר העקביות בין השניים

מחלקות כטיפוסי נתונים

מחלקות כטיפוסי נתונים

- ביסודה של גישת התכנות מונחה העצמים קיימת 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 הגישה לעצמים היא באמצעות טיפוסי הפניה לעצם
  - לא ניתן לגשת לעצם עצמו.
- כל מופע עשוי להכיל:
  - נתונים (data members, instance fields)
  - שרותים (instance methods)
  - פונקציות אתחול (בנאים, constructors)

מחלקות ועצמים

- כבר ראינו בקורס שימוש בטיפוסים שאינם פרימיטיביים: מחרוזת ומערך
  - גם ראינו שעקב שכיחות השימוש בהם יש להם הקלות תחביריות מסוימות (פטור מ- **new** והעמסת אופרטור +)

- ראינו כי עבודה עם טיפוסים אלה מערבת שתי ישויות נפרדות:
  - **העצם**: המכיל את המידע
  - **ההפנייה**: משתנה שדרכו ניתן לגשת לעצם

- זאת בשונה ממשתנים יסודיים (טיפוסים פרימיטיביים)
  - דוגמא:

```
int i =5 , j = 7;
String s = "Hello", t = "World";
```

i ו- **j** הם מופעים של **int** כשם ש "**hello**" ו- "**world**" הם מופעים של **String**. **s** ו **t** הם הפניות למחרוזות.

שרותי מופע

- למחלקות יש **שרותי מופע** – פונקציות אשר מופעלות על מופע מסוים של המחלקה

- תחביר של הפעלת שרות מופע הוא:

```
objRef.methodName(arguments)
```

לדוגמא:

```
String str = "SupercaliFrajalistic";
int len = str.length();
```

- זאת בשונה מזימון שרות מחלקה (static):

```
ClassName.methodName(arguments)
```

לדוגמא:

```
String.valueOf(15);       // returns the string "15"
```

- שימו של כי האופרטור נקודה (.) משמש בשני המקרים בתפקידים שונים לגמרי!

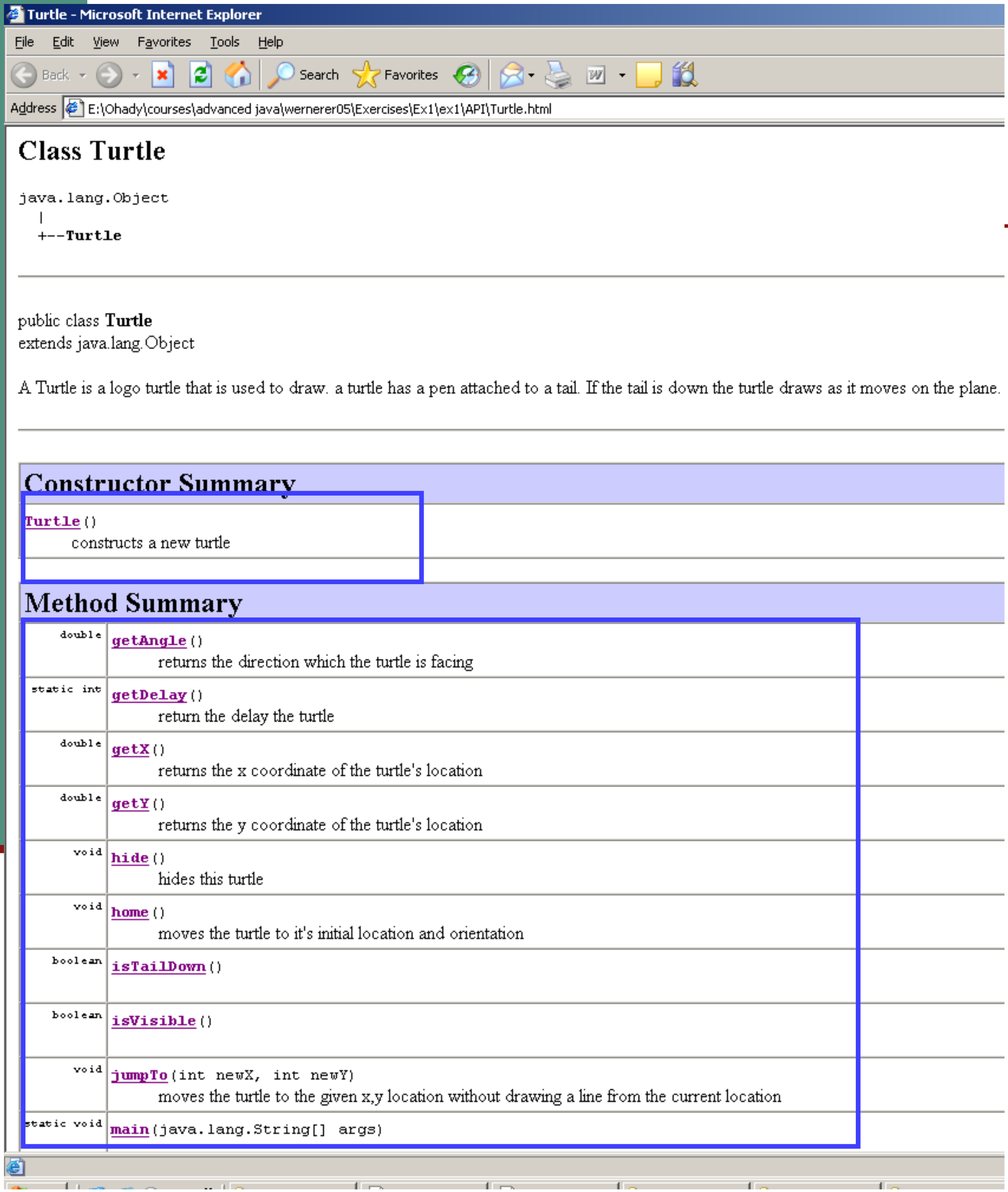
שימוש במחלקות קיימות

- 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



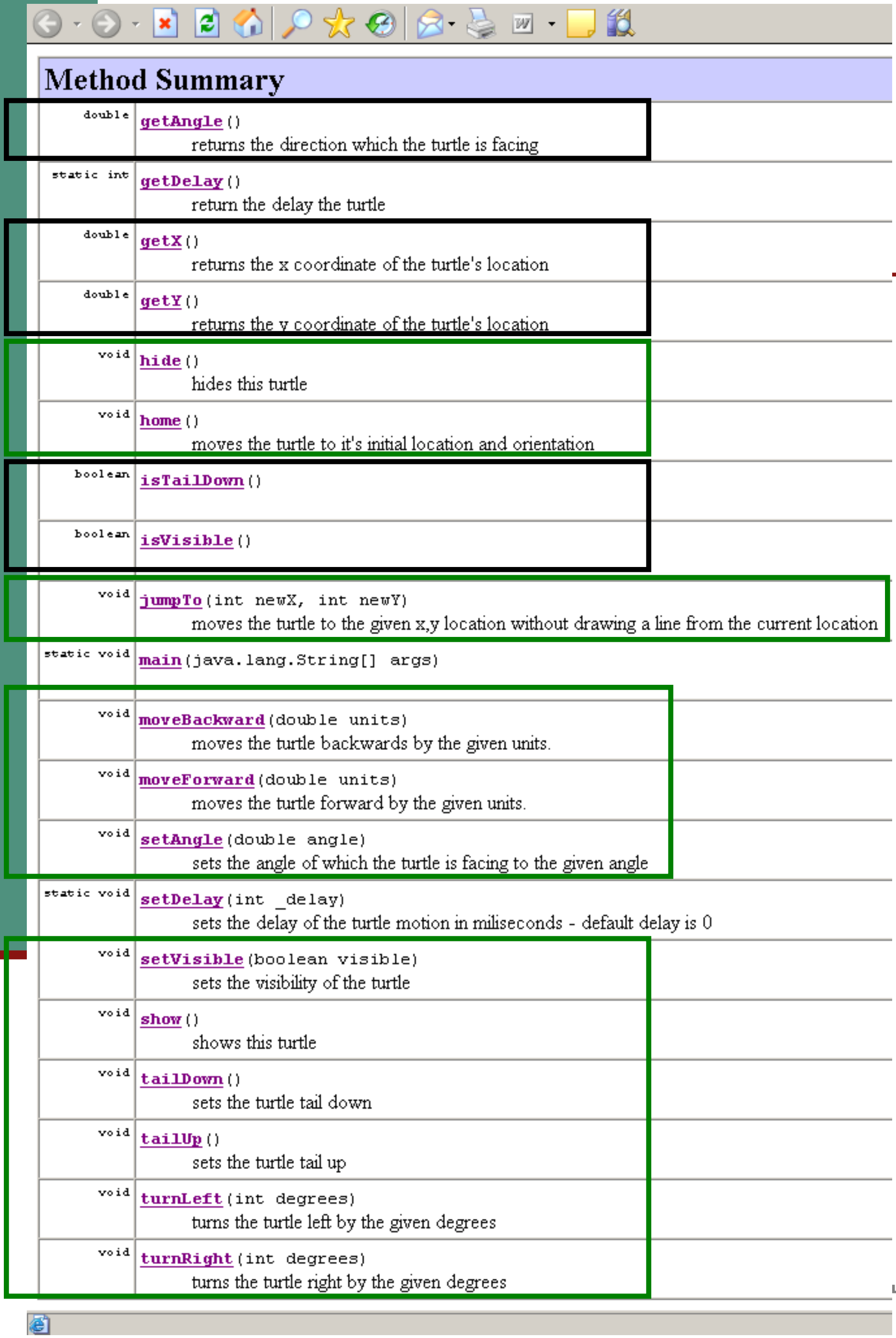
בנאי – פונקצית אתחול -
ניתן לייצר מופעים חדשים של
המחלקה ע"י קריאה לבנאי עם
האופרטור new

שרותים – נפריד בין 2 סוגים
שונים:

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

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

Turtle API



Method Summary

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

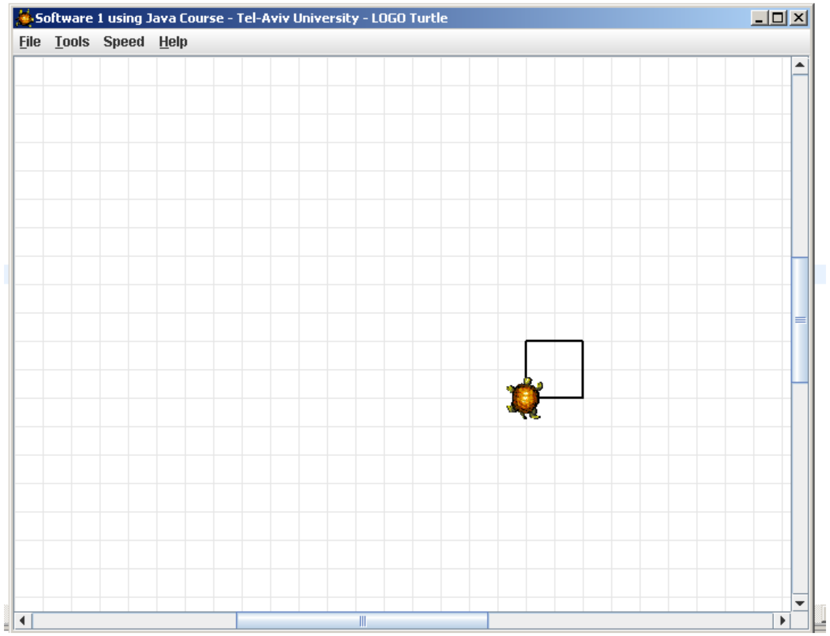
סוגים של שרותי מופע:

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

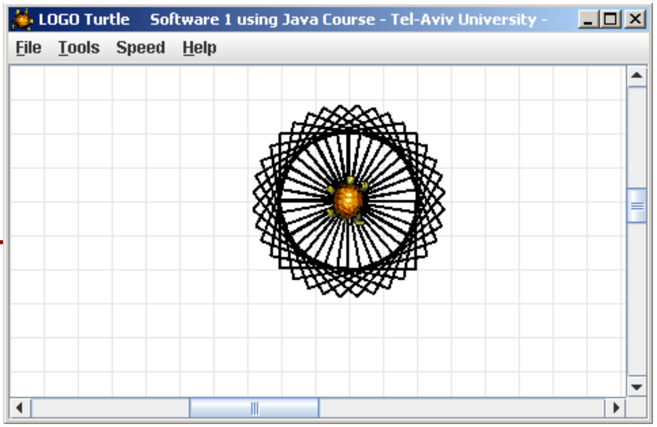
2. **פקודות** (commands) –
   - שרותים ללא ערך מוחזר
   - בדרך כלל משנים את מצב העצם שעליו הם פועלים

דוגמת שימוש

```java
public class TurleClient {

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

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

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

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

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

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



עוד דוגמת שימוש

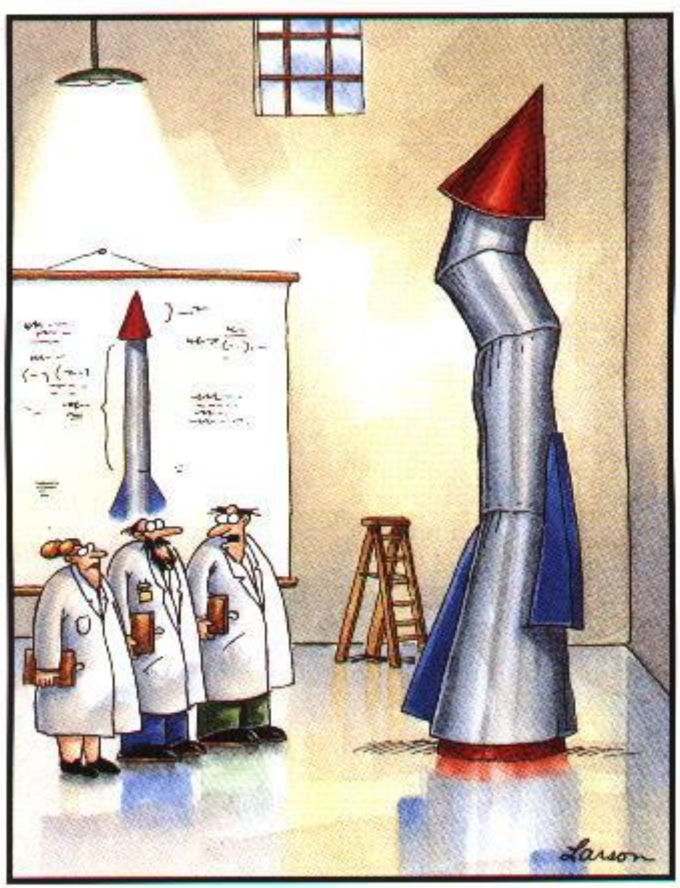



```java
public class TurleClient {

    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);
        }
    }
}
```

האם המחלקה צריכה להכיל כל שירות אפשרי?

- מדוע המחלקה **`Turtle`** לא הכילה מלכתחילה את השרותים **`drawSquare`** ו- **`drawSquarePattern`** ?
  - יש לכך יתרונות וחסרונות
- איך לימדנו את הצב את התעלולים החדשים?
- נשים לב להבדל בין השירותים הסטטיים שמקבלים **עצם כארגומנט** ומבצעים עליו פעולות ובין שרותי המופע אשר אינם מקבלים את העצם **כארגומנט מפורש** (העצם מועבר מאחורי הקלעים)

הגדרת טיפוסים חדשים



"It's time we face reality, my friends. ...
We're not exactly rocket scientists."

The cookie cutter

- כאשר מכינים עוגיות מקובל להשתמש בתבנית ברזל או פלסטיק כדי ליצור עוגיות בצורות מעניינות (כוכבים)

- תבנית העוגיות (cookie cutter) היא מעין **מחלקה** ליצירת עוגיות
- העוגיות עצמן הן **מופעים** (עצמים) שנוצקו מאותה תבנית

- כאשר ה JVM טוען לזכרון את קוד המחלקה עוד לא נוצר אף **מופע** של אותה המחלקה.

  המופעים יווצרו בזמן מאוחר יותר – כאשר הלקוח של המחלקה יקרא מפורשות לאופרטור `new`

- אם יש לנו תבנית לעוגיות, זה לא אומר שיש לנו עוגיות.
- התבנית מגדירה את הצורה של העוגיות, אבל לא את הטעם שלהן (וניל? שוקולד?)

דוגמא

- נתבון במחלקה **MyDate** לייצוג תאריכים:

```
public class MyDate {
    int day;
    int month;
    int year;
}
```

- שימו לב! המשתנים **day**, **month** ו- **year** הוגדרו ללא המציין **static** ולכן בכל מופע עתידי של עצם מהמחלקה **MyDate** יופיעו השדות האלה

- <u>שאלה</u>: כאשר ה JVM טוען לזיכרון את המחלקה איפה בזיכרון נמצאים השדות **day**, **month** ו- **year**?
- <u>תשובה</u>: הם עוד לא נמצאים! הם יווצרו רק כאשר לקוח ייצר מופע (עצם, אובייקט) מהמחלקה

לקוח של המחלקה MyDate

- **לקוח של המחלקה** הוא קטע קוד המשתמש ב- **MyDate**
- למשל: כנראה שמי שכותב יישום של יומן פגישות צריך להשתמש במחלקה
- דוגמא:

```
public class MyDateClient {

    public static void main(String[] args) {
        MyDate d1 = new MyDate();

        d1.day = 29;
        d1.month = 2;
        d1.year = 1984;

        System.out.println(d1.day + "/" + d1.month + "/" + d1.year);
    }
}
```

- בדוגמא אנו רואים:
  - שימוש באופרטור ה **new** ליצירת מופע חדש מטיפוס **MyDate**
  - שימוש באופרטור הנקודה לגישה לשדה של המופע המוצבע ע"י **d1**

אם שרות, אז עד הסוף

- האם התאריך d1 מייצג תאריך תקין?
- מה יעשה כותב היומן כאשר יצטרך להזיז את הפגישה בשבוע?
  - **האם `d1.day += 7` ?**
- כמו כן, אם למחלקה כמה לקוחות שונים – אזי הלוגיקה הזו תהיה משוכפלת אצל כל אחד מהלקוחות
- אחריותו של מי לוודא את תקינות התאריכים ולממש את הלוגיקה הנלווית?
- המחלקה היא גם מודול. אחריותו של הספק – כותב המחלקה – לממש את כל הלוגיקה הנלווית לייצוג תאריכים
  - כדי לאכוף את עקביות המימוש (*משתמר המחלקה*) על משתני המופע להיות פרטיים

בהמשך ניראה מימוש אחר של השירותים של MyData.
במימוש זה, לא נצטרך לשלוח את d כפרמטר לשירותים.

```
public class MyDate {

    private int day;
    private int month;
    private int year;

    public static void incrementDate(MyDate d) {
       // changes d to be the consequent day
    }

    public static String toString(MyDate d) {
       return d.day + "/" + d.month + "/" + d.year;
    }

    public static void setDay(MyDate d, int day) {
        /* changes the day part of d to be day if
         * the resulting date is legal */
    }

    public static int getDay(MyDate d) {
       return d.day;
    }

    private static boolean isLegal(MyDate d) {
       // returns if d represents a legal date
    }

    // more...
}
```

נראות פרטית

- מכיוון שהשדות **month**, **day** ו- **year** הוגדרו בנראות פרטית (**private**) לא ניתן להשתמש בהם מחוץ למחלקה (שגיאת קומפילציה)

```
public class MyDateClient {

    public static void main(String[] args) {
        MyDate d1 = new MyDate();

        d1.day = 29;
        d1.month = 2;
        d1.year = 1984;
    }
}
```

- כדי לשנות את ערכם יש להשתמש בשרותים הציבוריים שהוגדרו לשם כך

לקוח של המחלקה MyDate

```
public class MyDateClient {

    public static void main(String[] args) {
        MyDate d1 = new MyDate();

        MyDate.setDay(d1, 29);
        MyDate.setMonth(d1, 2);
        MyDate.setYear(d1, 1984);

        System.out.println(MyDate.toString(d1));
    }

}
```

כעת הדוגמא מתקמפלת אך עדיין נותרו בה שתי בעיות:

- השימוש בפונקציות גלובליות (סטטיות) מסורבל
  - עבור כל פונקציה אנו צריכים להעביר את **d1** כארגומנט
- מיד לאחר השימוש באופרטור ה **new** קיבלנו עצם במצב לא עיקבי
  - עד לביצוע השמת התאריכים הוא מייצג את התאריך הלא חוקי 0/0/00

שרותי מופע

- כדי לפתור את הבעיה הראשונה, נשתמש בסוג שני של שרותים הקיים ב Java – **שרותי מופע**
- שירותי מופע הם שרותים המשויכים <u>למופע מסוים</u> – הפעלה שלהם נחשבת כבקשה או שאלה מעצם מסוים – והיא מתבצעת בעזרת אופרטור הנקודה
- בגלל שהבקשה היא מעצם מסוים, אין צורך להעביר אותו כארגומנט לפונקציה
- מאחורי הקלעים הקומפיילר מייצר משתנה בשם **`this`** ומעביר אותו לפונקציה, ממש כאילו העביר אותו המשתמש בעצמו

ממתקים להמונים

- ניתן לראות בשרותי מופע **סוכר תחבירי** (syntactic sugar) לשרותי מחלקה

- ניתן לדמיין את שרות המופע `m()` של מחלקה C כאילו היה שרות מחלקה (סטטי) המקבל עצם מהטיפוס C כארגומנט:

```
public class C {

    public void m(args){
        ...
    }
}
```

```
public static void m(C thisObj, args) {
    ...
}
```

ממתקים להמונים

- בראייה זו, הקריאות למתודה `m()` של לקוחות המחלקה `C` יתורגמו ע"י העברת ההפניה שעליה בוצעה הקריאה כארגומנט לשרות הסטטי:

```
public class SomeClient {

   public static void main(String[] args) {
      C obj = new C();
      obj.m(args);
   }
}
```

```
C.m(obj,args)
```

"לא מה שחשבת"

- שרותי מופע מספקים תכונה נוספת ל Java פרט לסוכר התחבירי
- בהמשך הקורס נראה כי לשרותי המופע ב Java תפקיד מרכזי **בשיגור שרותים דינאמי** (dynamic dispatch), תכונה בשפה המאפשרת החלפת המימוש בזמן ריצה ופולימורפיזם
- תאור שרותי מופע כסוכר תחבירי הוא פשטני (**ושגוי!**) אך נותן **אינטואיציה טובה** לגבי פעולת השרות בשלב זה של הקורס

```java
public class MyDate {

    private int day;
    private int month;
    private int year;

    public          void incrementDate(          ){
        // changes itself to be the consequent day
    }

    public          String toString(          ){
        return this.day + "/" + this.month + "/" + this.year;
    }

    public          void setDay(           int day){
        /* changes the day part of itself to be day if
         * the resulting date is legal */
    }

    public          int getDay(          ){
        return this.day;
    }

    private          boolean isLegal(          ){
        // returns if the argument represents a legal date
    }

    // more...
}
```

הקוד הזה חוקי !

המשתנה **this** מוכר בתוך שרותי המופע כאילו הועבר ע"י המשתמש.

אולם לא חובה להשתמש בו

```java
public class MyDate {

    private int day;
    private int month;
    private int year;

    public  void incrementDate(){
        // changes current object to be the consequent day
    }

    public  String toString(){
       return day + "/" + month + "/" + year;
    }

    public  void setDay(int day){
        /* changes the day part of the current object to be day if
         * the resulting date is legal */
    }

    public  int getDay(){
       return day;
    }

    private  boolean isLegal(){
        // returns if the current object represents a legal date
    }

    // more...
}
```

(constructors) בנאים

- כדי לפתור את הבעיה שהעצם אינו מכיל ערך תקין מיד עם יצירתו נגדיר עבור המחלקה **בנאי**
- בנאי הוא **פונקצית אתחול** הנקראת ע"י אופרטור ה `new` מיד אחרי שהוקצה מקום לעצם החדש. שמה כשם המחלקה שהיא מאתחלת וחתימתה אינה כוללת ערך מוחזר
- המוטיבציה המרכזית להגדרת בנאים היא יצירת עצם שהוא עקבי עם השימוש המיועד שלו (בהמשך נדבר על *משתמר מחלקה* ומצב *מופשט בעל משמעות)*
  - למשל, נרצה שהאובייקט ה **`MyDate`** שאנחנו מייצרים יכיל תאריך חוקי מיד עם יצירתו

```java
public class MyDate {

    public MyDate(int day, int month, int year) {
        this.day = day;
        this.month = month;
        this.year = year;
    }

    // ...
}
```

הגדרת בנאי ל MyDate

```java
public class MyDateClient {

    public static void main(String[] args) {
        MyDate d1 = new MyDate(29,2,1984);
        d1.incrementDate();

        System.out.println(d1.toString());
    }
}
```

קוד לקוח המשתמש ב- MyDate

בנאים

האם ניתן לוותר על השימוש ב **this** בבנאי שהגדרנו?

```
public class MyDate {

    public MyDate(int day, int month, int year) {
        this.day = day;
        this.month = month;
        this.year = year;
    }

}
```

כעת מופיעה בקוד ההשמה הבאה:

day=day;

בגלל ששם השדה <u>זהה</u> לשם הפרמטר, הורדת השימוש ב **this** מייצרת השמה חסרת משמעות אשר אינה מאתחלת את השדה **day**.

בנאי ברירת מחדל

- במידה ולא הוגדר אף בנאי למחלקה, נוצר בנאי ברירת מחדל (default constructor).
- בנאי ברירת המחדל מתנהג בדיוק כמו הבנאי הבא:

```
public class MyDate {

    public MyDate() {
    }
}
```

- ומאפשר יצירה של אובייקט מטיפוס **MyDate** באופן הבא:

```
public static void main(String[] args) {
    MyDate d1 = new MyDate();
}
```

בנאים

- לאילו ערכים מאותחלים שדות מחלקה שלא אותחלו בבנאי?

שדות של מחלקות מאותחלים אוטומטית לערכים הדיפולטיים של כל טיפוס (0, null, false), כך שאין חובה לאתחל ערכים אלה בבנאי.

- זכרון שמוקצה על ה Heap מאותחל אוטומטית.

בנאים

- האם הקוד הבא יתקמפל?

```
public class MyDate {

    public MyDate(int day, int month, int year) {
        this.day = day;
        this.month = month;
        this.year = year;
    }

    public static void main(String[] args) {
        MyDate d1 = new MyDate();
    }

}
```

בנאי ברירת מחדל נוצר <u>רק כאשר לא הוגדר אף בנאי אחר במחלקה</u>.
אם קיים מימוש של בנאי כלשהו, הבנאי הריק לא נוצר אוטומטית ויש לממש אותו בקוד במידה ונרצה להשתמש בו.

מודל הזיכרון של זימון שרותי מופע

מודל הזיכרון של זימון שרותי מופע

- בדוגמא הבאה נראה כיצד מייצר הקומפיילר עבורנו את ההפניה **`this`** עבור כל בנאי וכל שרות מופע

- נתבונן במחלקה **`Point`** המייצגת נקודה במישור הדו מימדי. כמו כן המחלקה מנהלת מעקב בעזרת משתנה גלובלי (סטטי) אחר מספר העצמים שנוצרו מהמחלקה

- בהמשך הקורס נציג מימוש מלא ומעניין יותר של המחלקה, אולם כעת לצורך פשטות הדוגמא נסתפק בבנאי, שדה מחלקה, שני שדות מופע ושלושה שרותי מופע

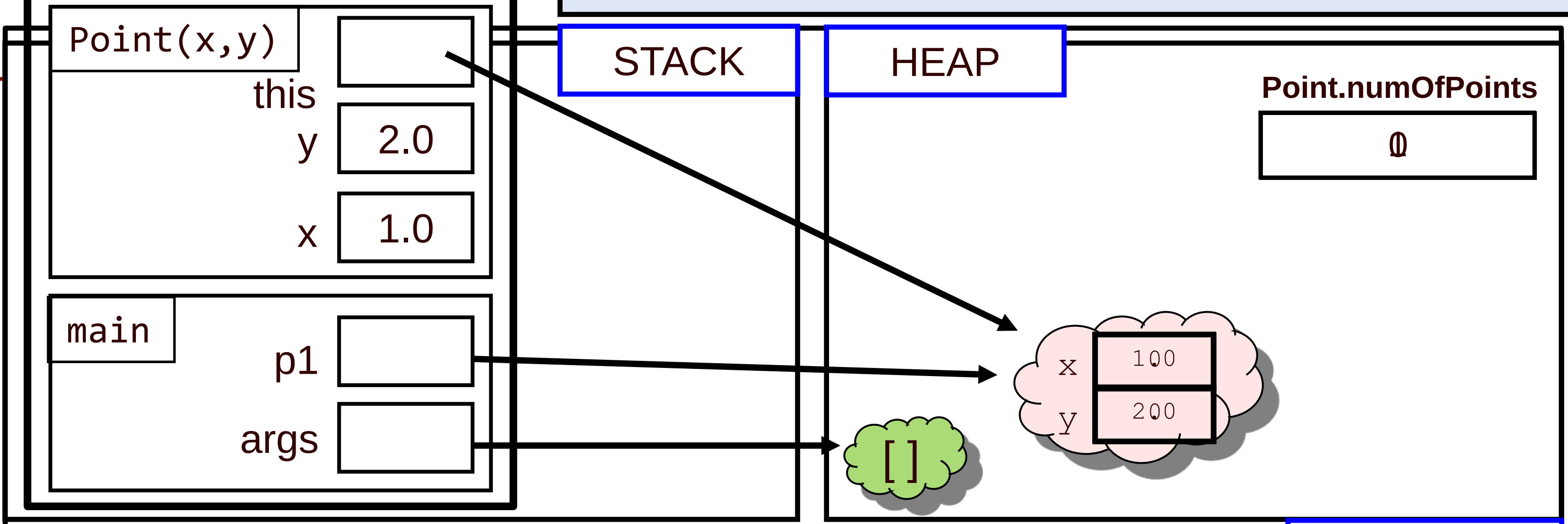
```java
public class Point {

    private static double numOfPoints;

    private double x;
    private double y;

    public Point(double x, double y){
       this.x = x;
       this.y = y;
       numOfPoints++;
    }

    public double getX() {
       return x;
    }

    public void setX(double newX) {
       if(newX > 0.0 && newX < 100.0)
          doSetX(newX);
    }

    public void doSetX(double newX) {
       x = newX;
    }

    // More methods...
}
```

PointUser

```
public class PointUser {

    public static void main(String[] args) {
        Point p1 = new Point(1.0, 2.0);
        Point p2 = new Point(10.0, 20.0);

        p1.setX(11.0);
        p2.setX(21.0);

        System.out.println("p1.x == " + p1.getX());
    }

}
```

בכל הפעלה של שרות מופע או בנאי יש עצם מטרה (target) שעליו הופעל השרות. ההפניה this מצביעה לעצם זה

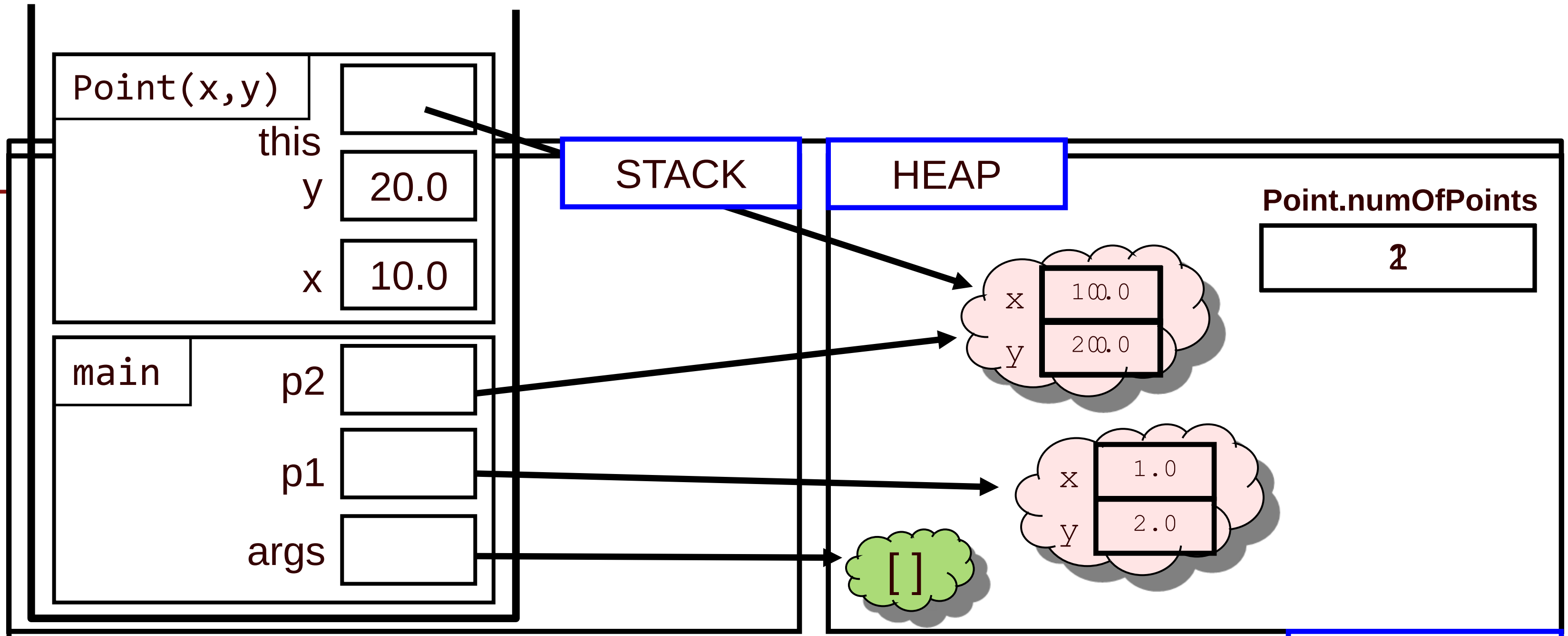
במהלך ריצת בנאי, ההפניה this מצביעה על העצם שזה עתה הוקצה



CODE

```java
public class PointUser {

    public static void main(String[] args) {
        Point p1 = new Point(1.0, 2.0);
        Point p2 = new Point(10.0, 20.0);

        p1.setX(11.0);
        p2.setX(21.0);

        System.out.println("p1.x == "
                            + p1.getX());
    }
}
```

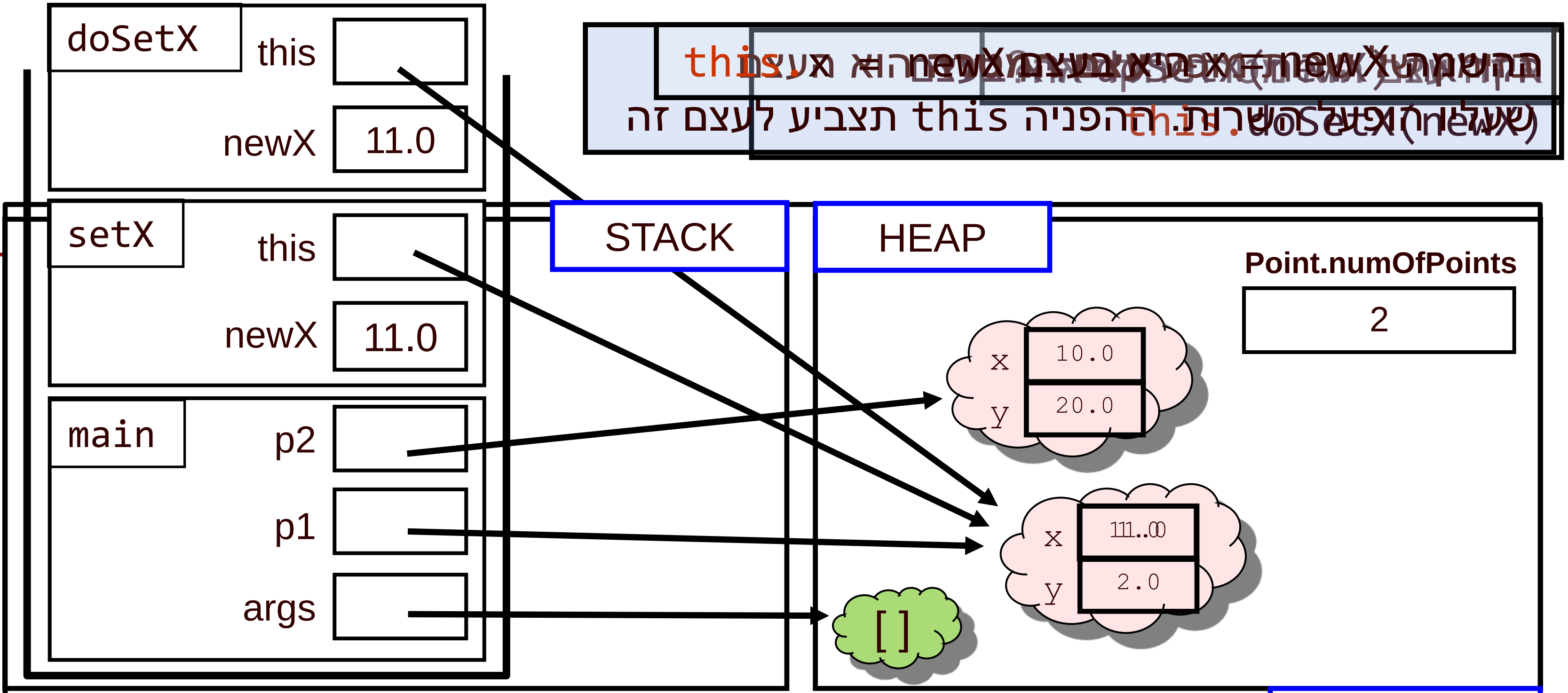
```java
public class Point {
    public Point(double x, double y){
        this.x = x;
        this.y = y;
        numOfPoints++;
    }

    public double getX()
    { return x; }

    public void setX(double newX) {
        if(newX > 0.0 && newX < 100.0)
            doSetX(newX);
    }

    public void doSetX(double newX)
    { x = newX; }
```



CODE

```
public class PointUser {

    public static void main(String[] args) {
        Point p1 = new Point(1.0, 2.0);
        Point p2 = new Point(10.0, 20.0);

        p1.setX(11.0);
        p2.setX(21.0);

        System.out.println("p1.x == "
                                + p1.getX());
    }
}
```

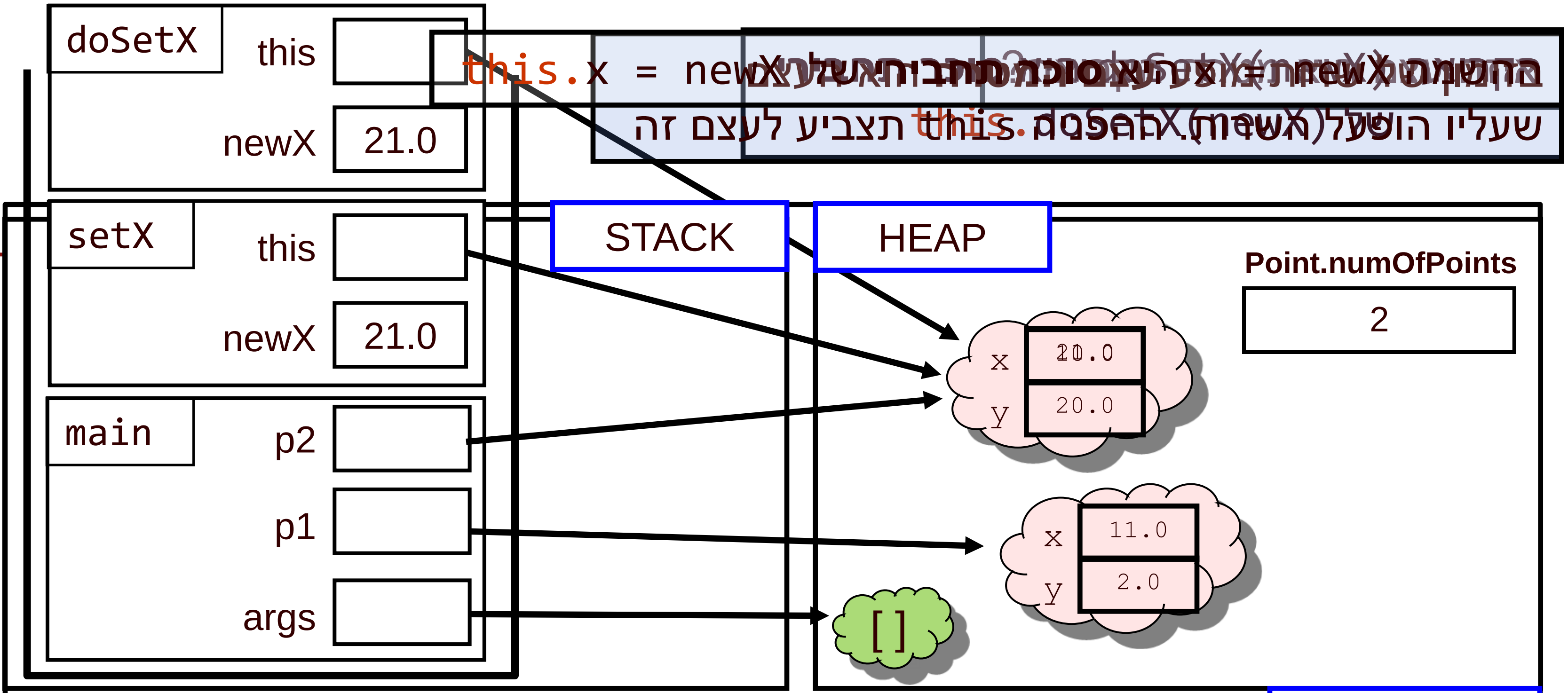
```
public class Point {
    public Point(double x, double y){
        this.x = x;
        this.y = y;
        numOfPoints++;
    }

    public double getX()
    { return x; }

    public void setX(double newX) {
        if(newX > 0.0 && newX < 100.0)
            doSetX(newX);
    }

    public void doSetX(double newX)
    { x = newX; }
```



CODE

```java
public class PointUser {

    public static void main(String[] args) {
        Point p1 = new Point(1.0, 2.0);
        Point p2 = new Point(10.0, 20.0);

        p1.setX(11.0);
        p2.setX(21.0);

        System.out.println("p1.x == "
                                + p1.getX());
    }

}
```

```java
public class Point {

    public Point(double x, double y){
        this.x = x;
        this.y = y;
        numOfPoints++;
    }

    public double getX()
    { return x; }

    public void setX(double newX) {
        if(newX > 0.0 && newX < 100.0)
                this.doSetX(newX);
    }

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

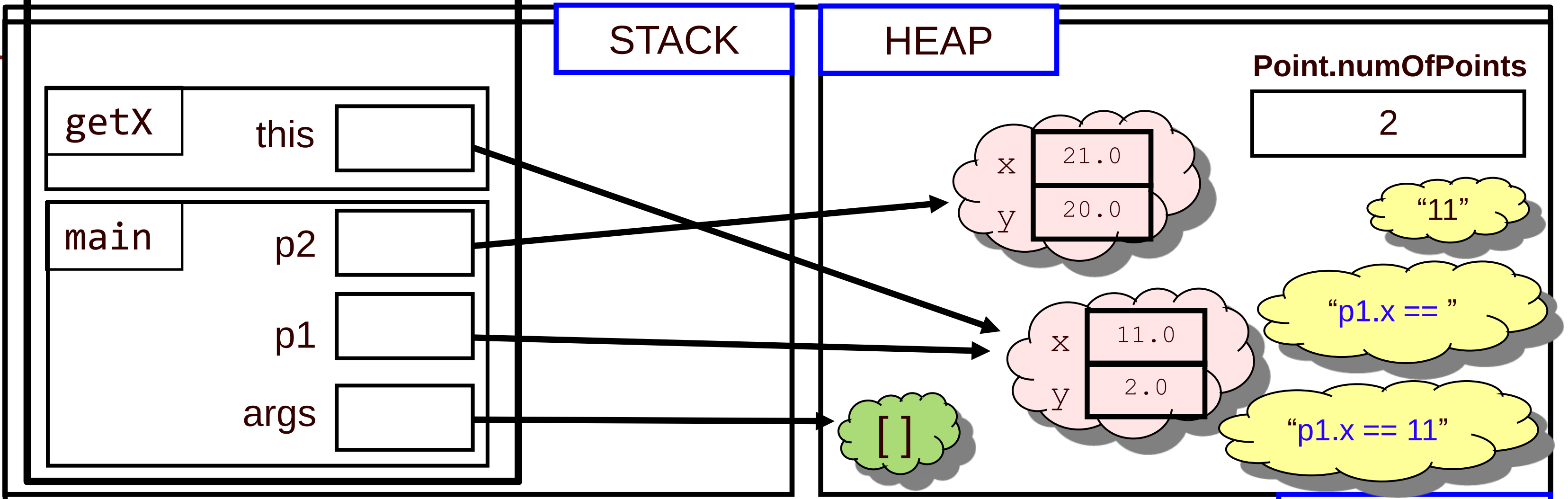


CODE

```
public class PointUser {

    public static void main(String[] args) {
        Point p1 = new Point(1.0, 2.0);
        Point p2 = new Point(10.0, 20.0);

        p1.setX(11.0);
        p2.setX(21.0);

        System.out.println("p1.x == "
                           + p1.getX());
    }

}
```

```
public class Point {

    public Point(double x, double y){
        this.x = x;
        this.y = y;
        numOfPoints++;
    }

    public double getX()
    { return x; }

    public void setX(double newX) {
        if(newX > 0.0 && newX < 100.0)
            this.doSetX(newX);
    }

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

המשפט "return this.x" הוא בעצם



CODE

```java
public class PointUser {

    public static void main(String[] args) {
        Point p1 = new Point(1.0, 2.0);
        Point p2 = new Point(10.0, 20.0);

        p1.setX(11.0);
        p2.setX(21.0);

        System.out.println("p1.x == "
                            + p1.getX());
    }

}
```

```java
public class Point {

    public Point(double x, double y){
        this.x = x;
        this.y = y;
        numOfPoints++;
    }

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

    public void setX(double newX) {
        if(newX > 0.0 && newX < 100.0)
            doSetX(newX);
    }

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

סיכום ביניים

- **שרותי מופע** (instance methods) בשונה משרותי מחלקה (static method) פועלים על עצם מסוים (this)
  - בעוד ששרותי מחלקה פועלים בדרך כלל על הארגומנטים שלהם
- **משתני מופע** (instance fields) בשונה ממשתני מחלקה (static fields) הם **שדות בתוך עצמים**. הם נוצרים רק כאשר נוצר עצם חדש מהמחלקה (ע"י new)
  - בעוד ששדות מחלקה הם משתנים גלובלים. קיים עותק אחד שלהם, שנוצר בעת טעינת קוד המחלקה לזכרון, ללא קשר ליצירת עצמים מאותה המחלקה