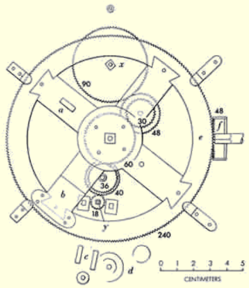
תוכנה 1 בשפת Java
שיעור מספר 3: מודל הזיכרון ושירותים

דן הלפרין

בית הספר למדעי המחשב
אוניברסיטת תל אביב

על סדר היום

- מודל הזיכרון של Java
  - Heap and Stack
  - העברת ארגומנטים
- מנגנוני שפת Java
- שרותים
  - הפשטה
  - חוזה של שרותים



העברת ארגומנטים

- כאשר מתבצעת קריאה לשרות, ערכי הארגומנטים נקשרים לפרמטרים הפורמלים 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.** אין העתקה של העצם שאליו מתייחסים – זאת בשונה משפות אחרות כגון C++ שבהם קיימת גם שיטת העברה by reference

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

HEAP

משתנים גלובלים ועצמים – אינו תלוי במתודה הנוכחית שרצה

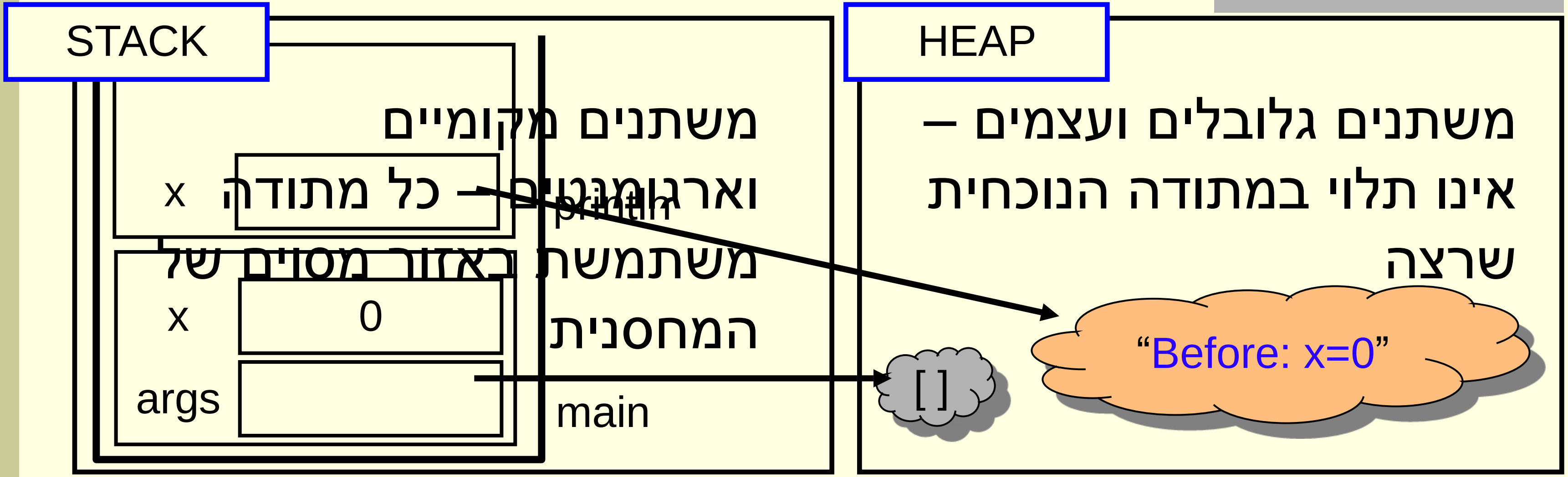
STACK

משתנים מקומיים וארגומנטים – כל מתודה משתמשת באזור מסוים של המחסנית

CODE

קוד התוכנית

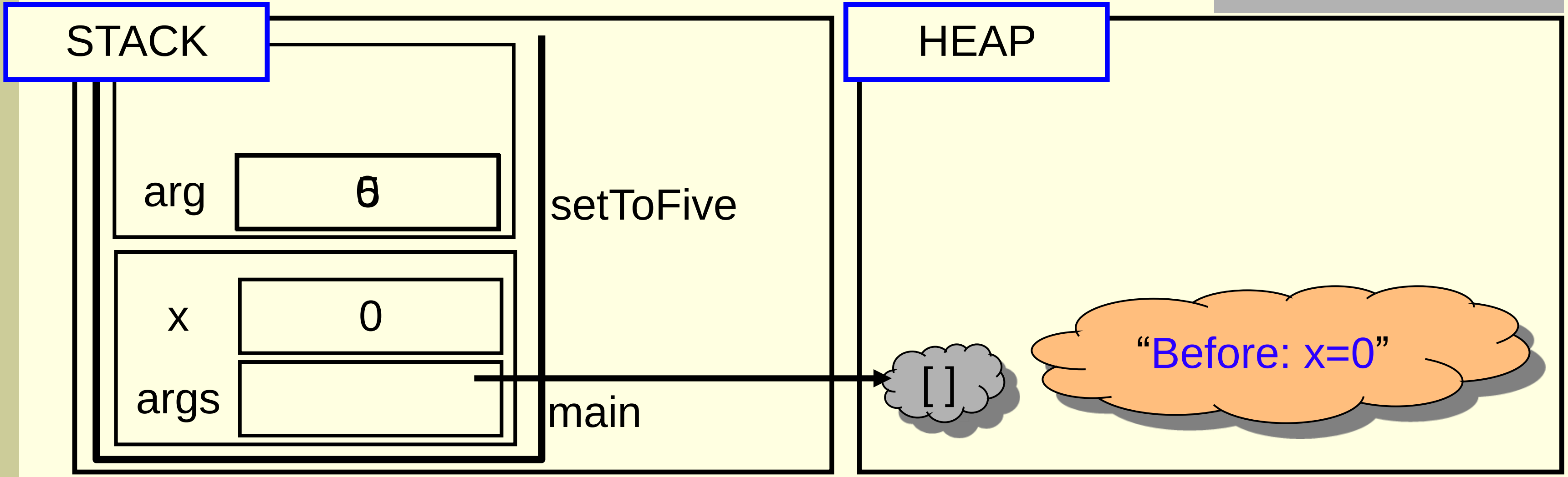
Primitives by value



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

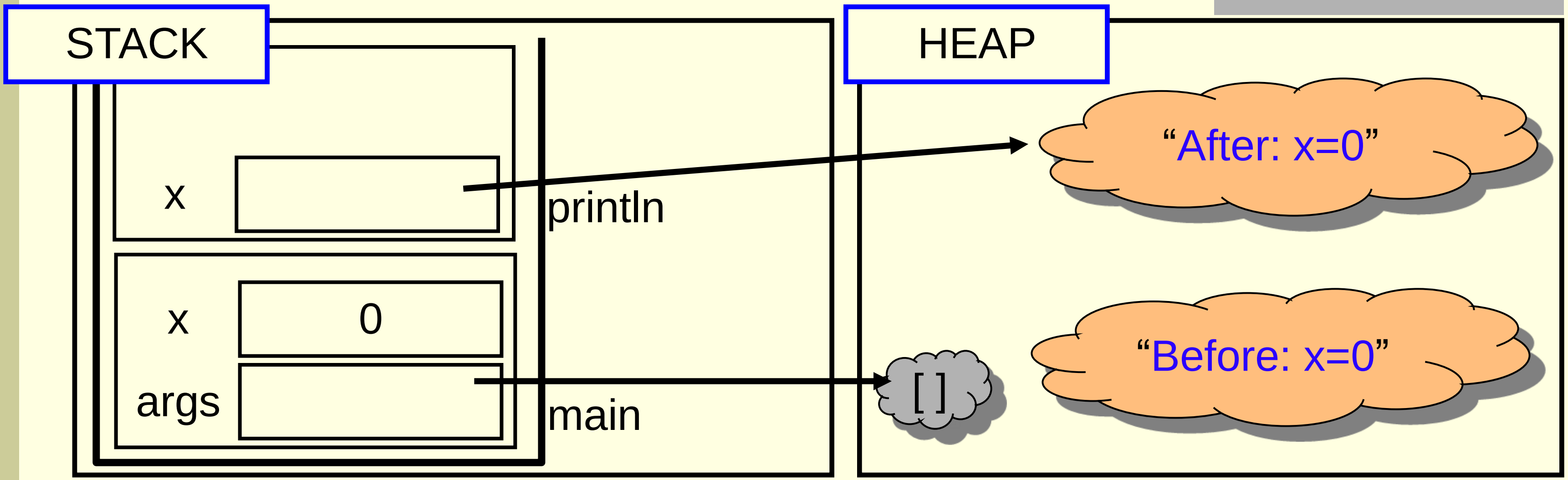


CODE

```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);
    }
}
```

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

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

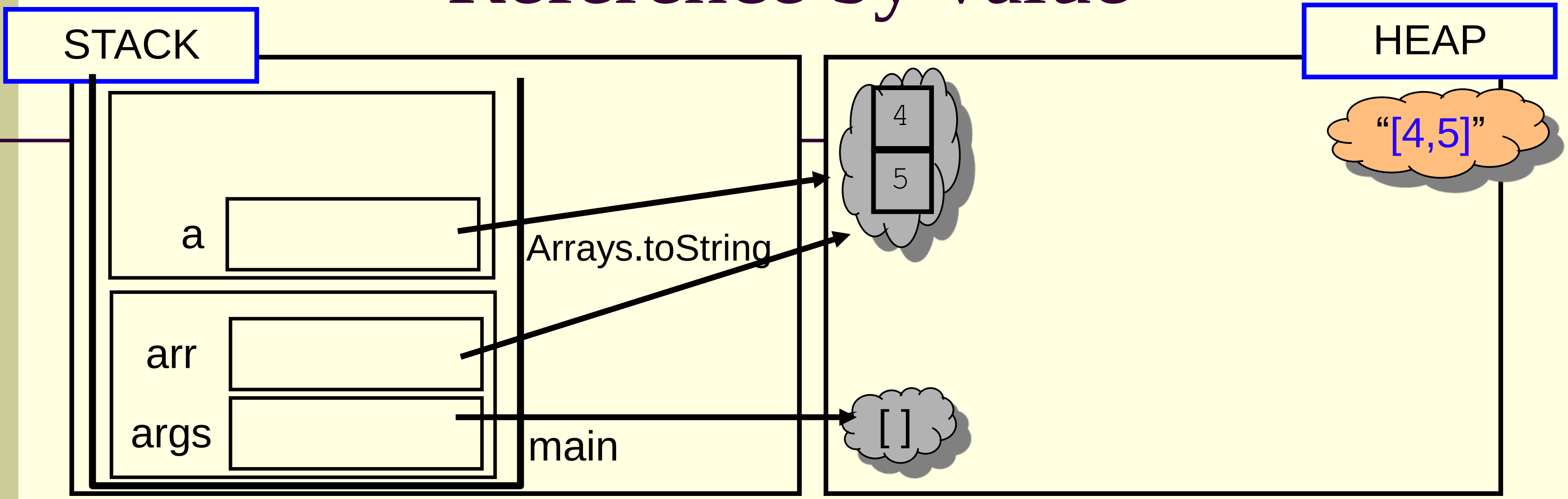
שמות מקומיים

- בדוגמא ראינו כי הפרמטר הפורמלי **arg** קיבל את הערך האקטואלי של הארגומנט x
- בחירת השמות השונים לא משמעותית - יכולנו לקרוא לשני המשתנים באותו שם ולקבל התנהגות זהה
- שם של משתנה מקומי **מסתיר** משתנים בשם זהה הנמצאים בתחום עוטף או גלובלים
- 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?

```java
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));
    }
}
```

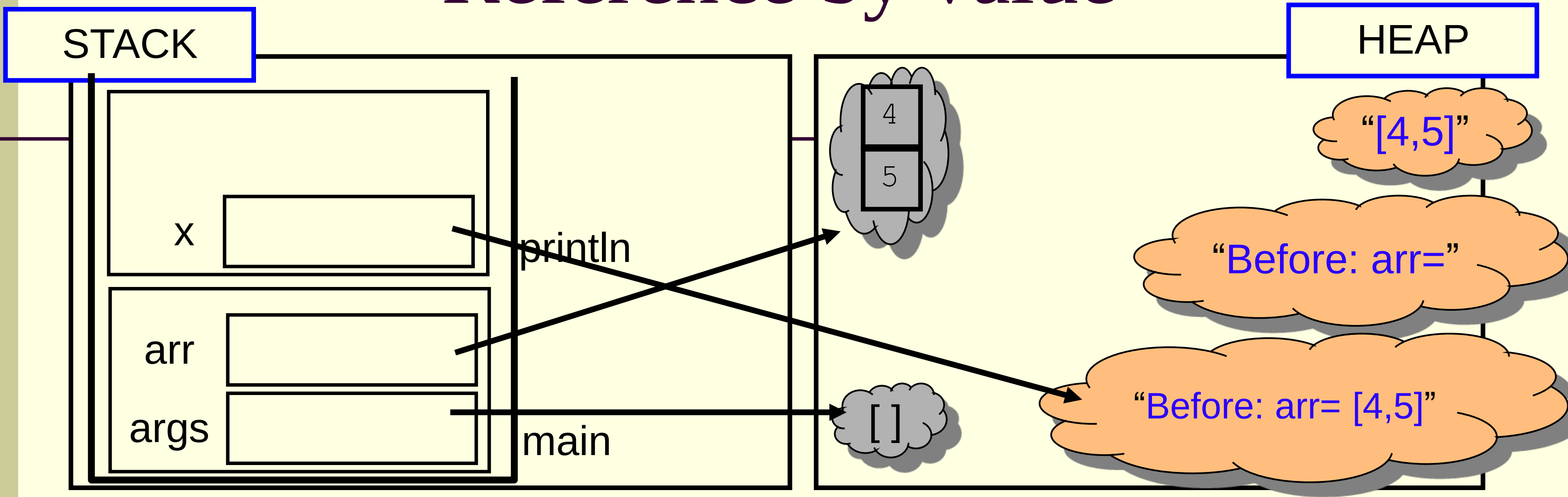
Reference by value



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));
    }
}
```

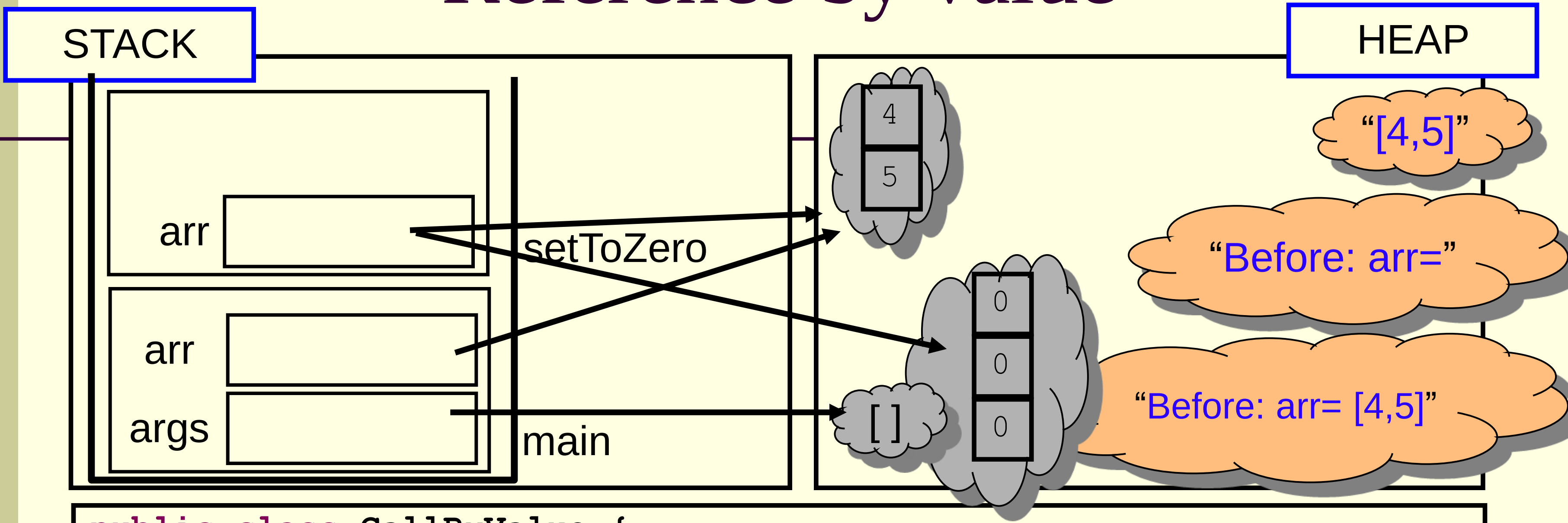
Reference by value



CODE

```
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));
    }
}
```

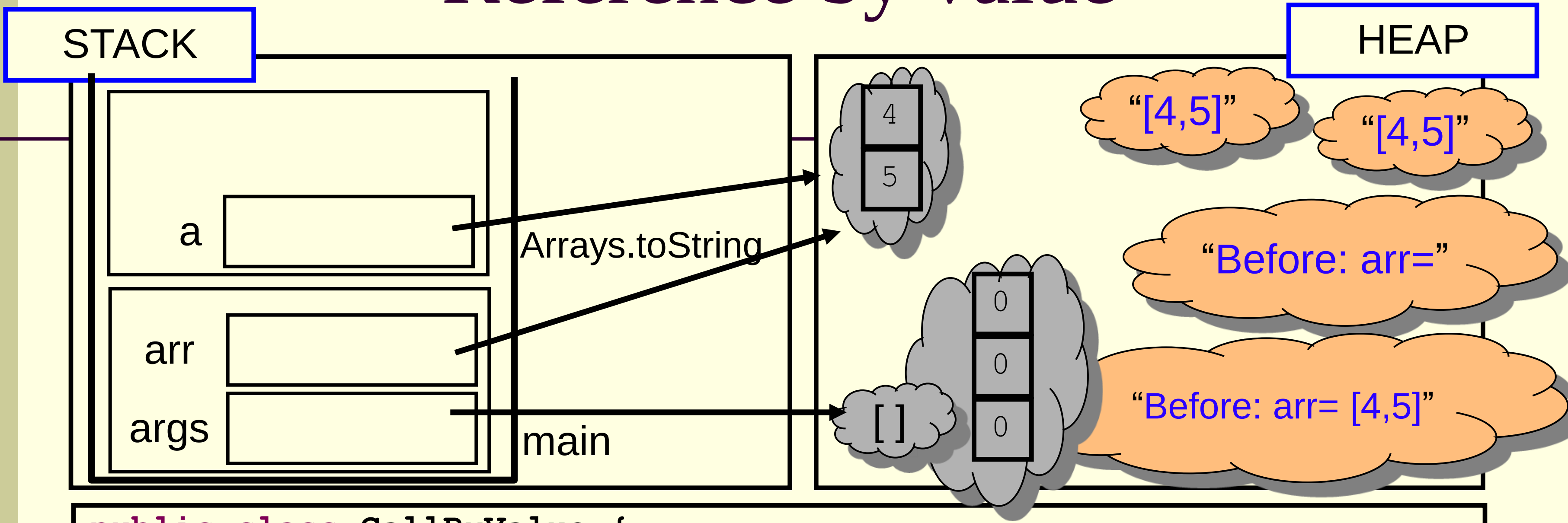
Reference by value



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));
    }
}
```

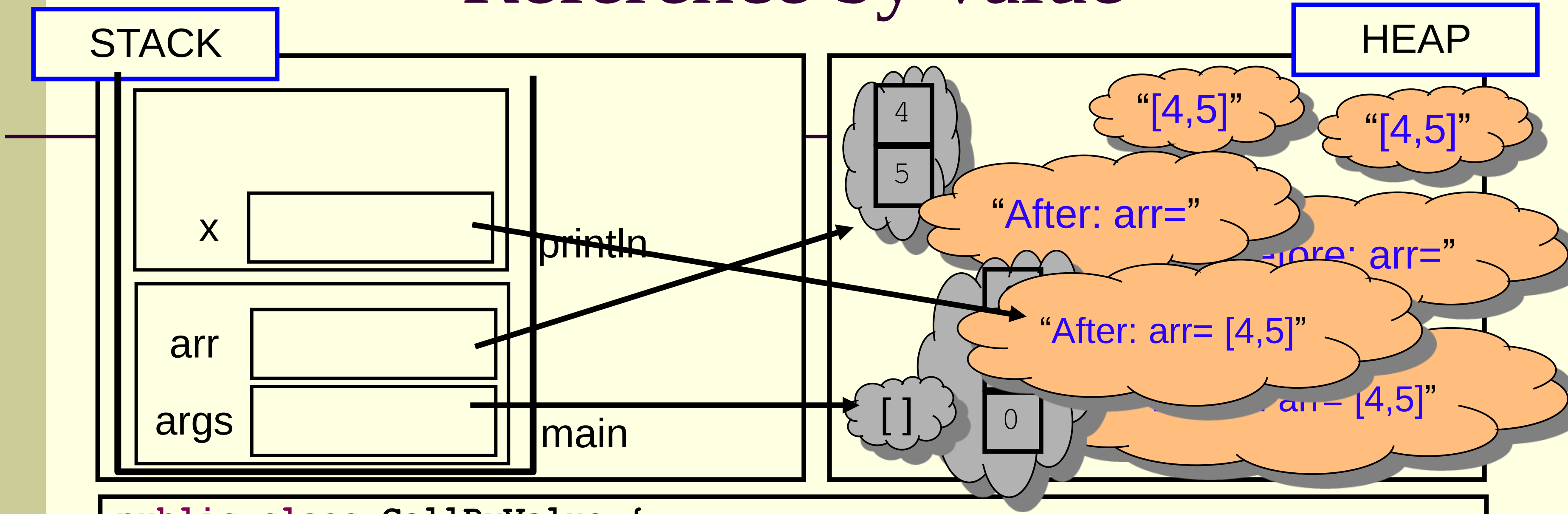
Reference by value



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));
    }
}
```

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

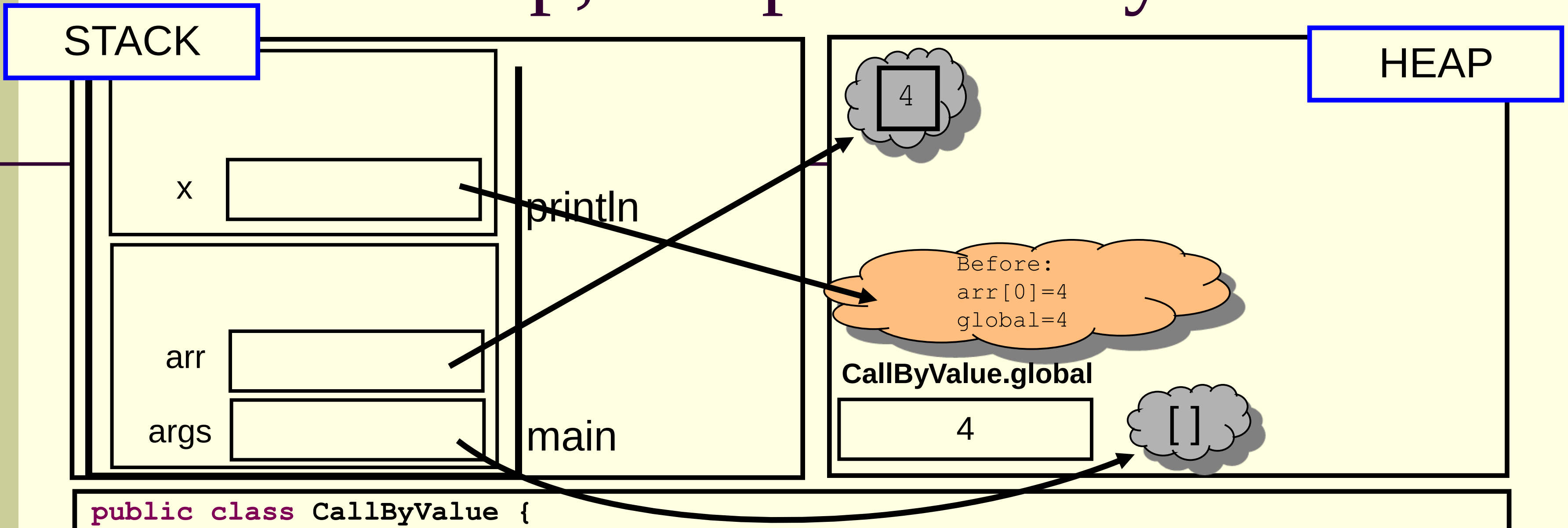
- אז איך יכולה מתודה להשפיע על ערכים במתודה שקראה לה?
  - ע"י ערך מוחזר
  - ע"י גישה למשתנים או עצמים שהוקצו ב- Heap

- 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: 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);
    }
}
```

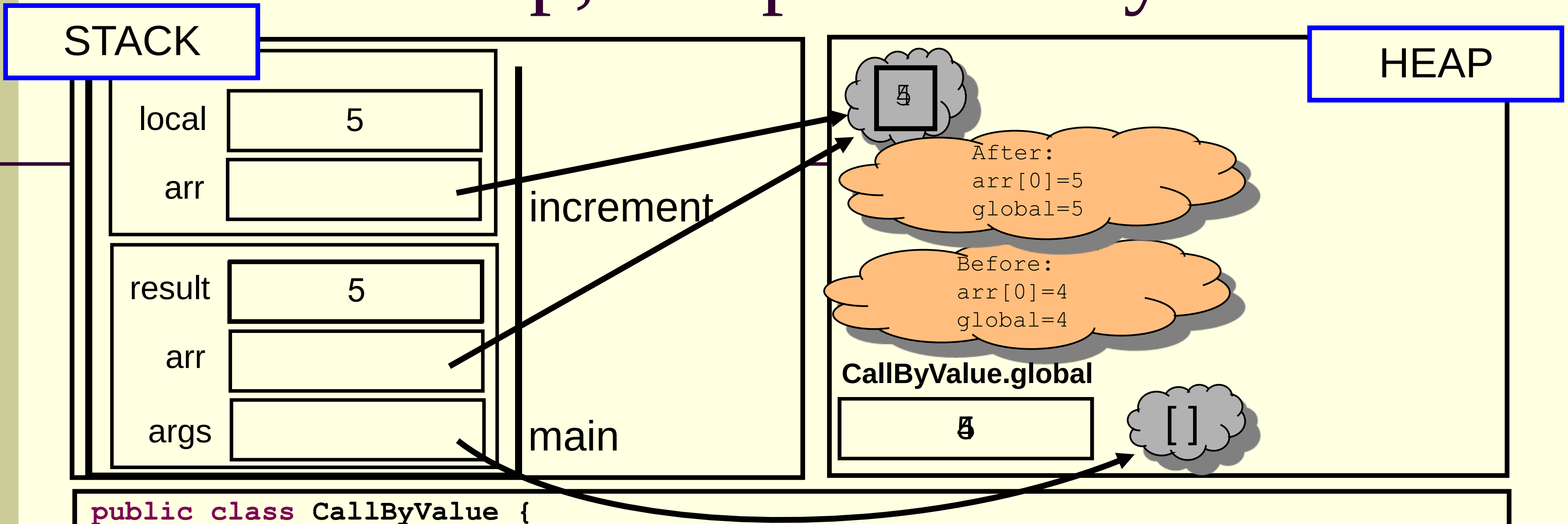
Heap, Heap – Hooray!



CODE

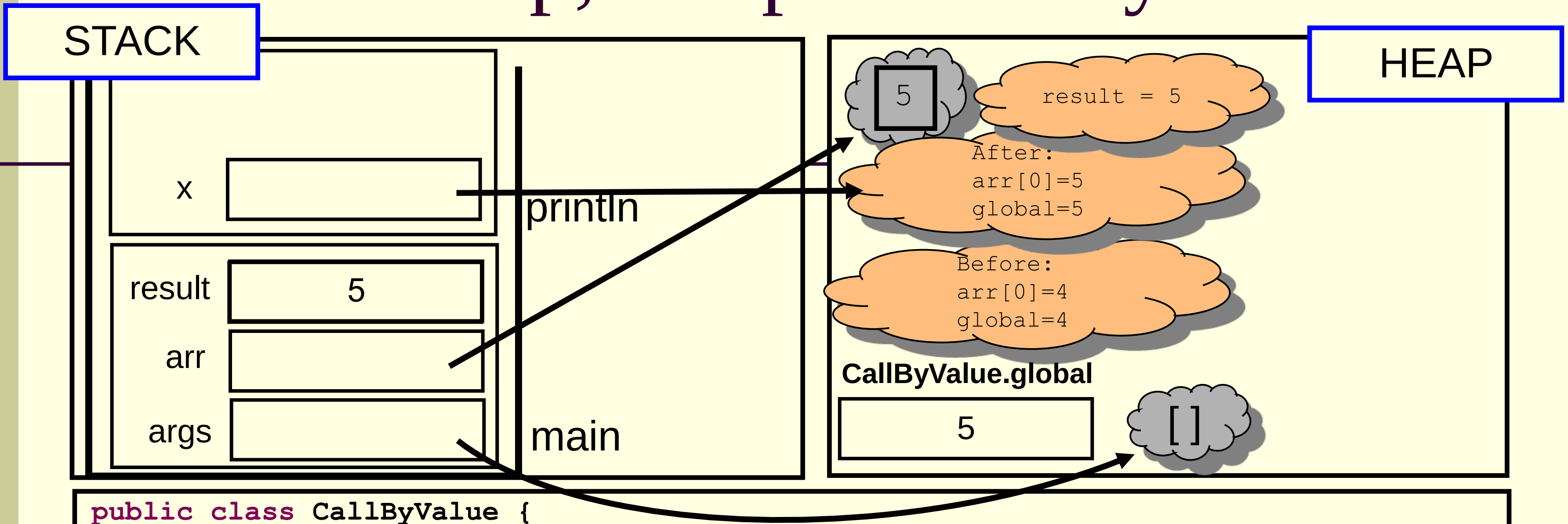
```
public class CallByValue {

    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);
    }
}
```

Heap, Heap – Hooray!




```
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);
    }
}
```

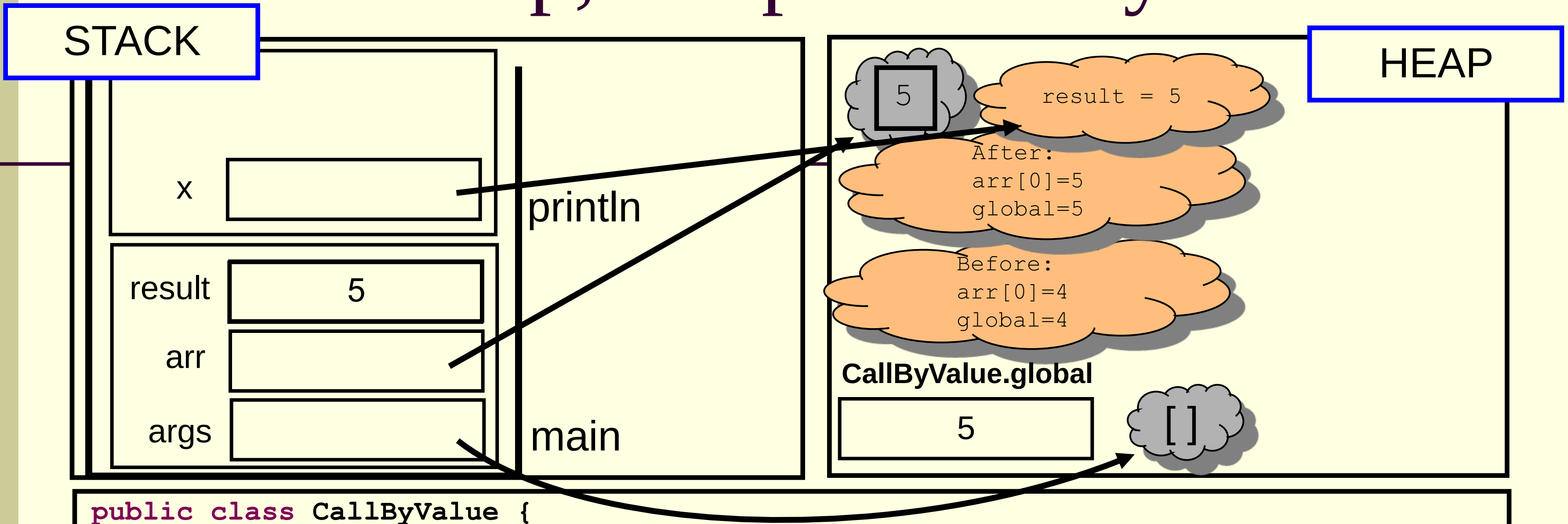
Heap, Heap – Hooray!




```
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);
    }
}
```

CODE

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)

- לכאורה אוקסימורון
- 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
- בשפות אחרות (למשל #C) קיים תחביר מיוחד לסוג כזה של ארגומנטים – ב Java אין לכך סימון מיוחד

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 – באתר הקורס

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"

מנגנוני שפת JAVA

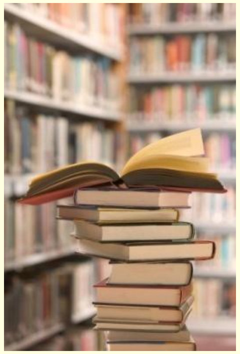
המחלקה כספריה של שרותים

- 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 הקורס

- גם בהן כבר ראינו אוסף שרותים (static) שימושיים לדוגמא:

- **`Integer`**
- **`Character`**
- **`String`**



המחלקה כספריה של שרותים

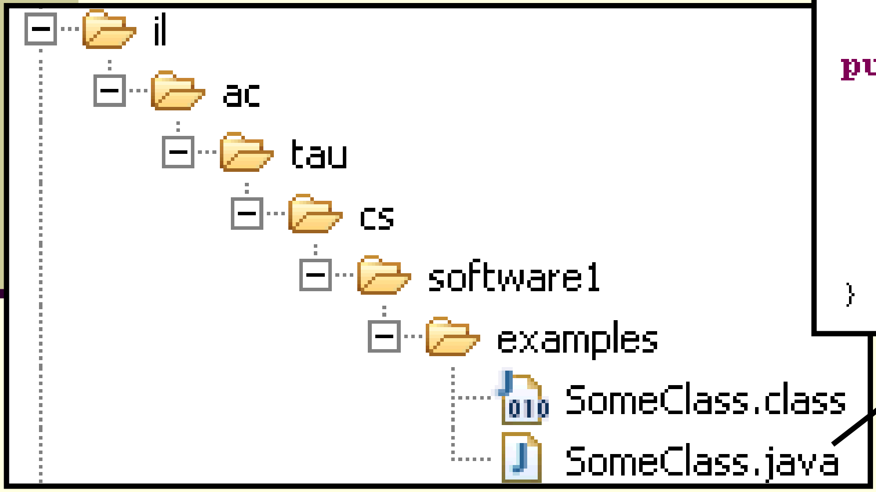
- ואולם קיימות ב- Java גם כמה מחלקות המשמשות כספריות בלבד. בין השימושיות שבהן:
  - **`java.lang.Math`**
  - **`java.util.Arrays`**
  - **`java.lang.System`**

חבילות ומרחב השמות

- מרחב השמות של Java היררכי
  - בדומה לשמות תחומים באינטרנט או שמות 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) {
        //...
    }
}
```

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`** (**J**ava **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

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

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

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

    /** Documetntaion 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;
    }
}
```

Java API

- חברת Sun תיעדה את כל מחלקות הספרייה של שפת Java וחוללה עבורן בעזרת **`javadoc`** אתר תיעוד מקיף ומלא הנמצא ברשת:

http://download.oracle.com/javase/7/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

שרותים

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 כבר ראינו 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

שרותים והפשטה
דוגמא

```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);
}
```

```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://martinfowler.com/refactoring/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 למתודה קטע קוד התלוי במשתנה מקומי

לקוח וספק במערכת תוכנה

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

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

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

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

לקוח וספק במערכת תוכנה

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

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

Supplier.java

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

Client.java

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

פערי הבנה

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

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

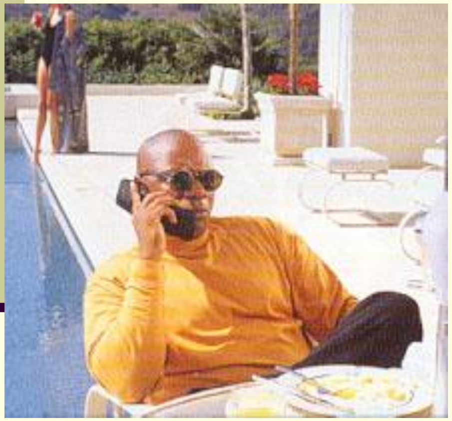
- לדוגמא: נתבון בשרות **divide** המקבל שני מספרים ומחזיר את המנה שלהם:

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

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

"Let us speak of the unspeakable"

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



יותר מדי קצוות פתוחים...

- אין בהכרח תשובה **נכונה** לגבי השאלות על הצורה שבה על divide לפעול

- ואולם יש לציין במפורש:
  - מה היו **ההנחות** שביצע כותב הפונקציה
    - במקרה זה הנחות על הארגומנטים (האם הם מתחלקים, אפס במכנה וכו')
  - מהי **התנהגות** הפונקציה במקרים השונים
    - בהתאם לכל המקרים שנכללו בהנחות

- פרוט ההנחות וההתנהגויות השונות מכונה **החוזה** של הפונקציה

- ממש כשם שבעולם העסקים נחתמים חוזים בין ספקים ולקוחות
  - קבלן ודיירים, מוכר וקונים, מלון ואורחים וכו'...

עיצוב על פי חוזה (design by contract)

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

חוזה אפשרי ל- divide

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

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

חוזה אפשרי אחר ל- divide

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

תנאי קדם סובלניים מסבכים את מימוש הפונקציה – כפי שמתבטא בחוזה

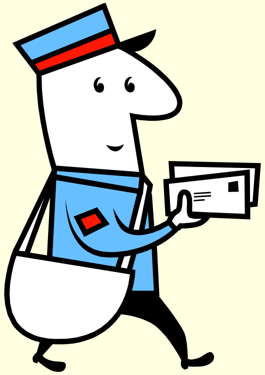
החוזה והמצב

- חוזה של שרות אינו כולל רק את הארגומנטים שלו
- תנאי קדם של חוזה יכול להגדיר **מצב** (תמונת זיכרון, קשירת ערכי משתנים) שרק בו ניתן לקרוא לפונקציה

- לדוגמא: במחלקה מסוימת קיימים שרות **המאתחל** מבנה נתונים ושרות **הקורא** מאותו מבנה נתונים (שדה מחלקה)
- תנאי הקדם של שרות הקריאה יכול להיות שמבנה הנתונים כבר אותחל ושנותרו בו הודעות
- נשים לב שמימוש `getNextMessage` מתעלם לחלוטין מהמקרים שבהם תנאי הקדם אינו מתקיים
- המימוש לא בודק את תנאי הקדם בגוף המתודה

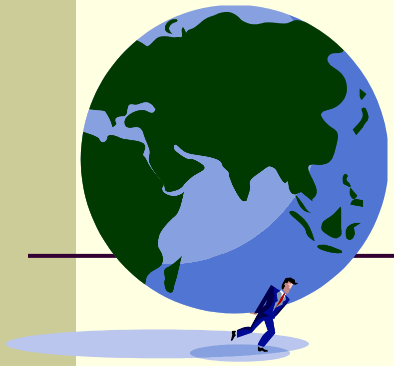
הדואר בא היום

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

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

/**
 * @pre isIntialized , "you must be logged in first"
 * @pre head < messages.length , "more messages to read"
 * @post head == $prev(head)+1 , "increments head"
 */
public static String getNextMessage(){
    return messages[head++];
}
```



שרות לעולם לא יבדוק את תנאי הקדם שלו

- שרות לעולם לא יבדוק את תנאי הקדם שלו
- גם לא "ליתר ביטחון"
- אם שרות בודק תנאי קדם ופועל לפי תוצאת הבדיקה, אזי יש לו התנהגות מוגדרת היטב עבור אותו תנאי – כלומר הוא אינו תנאי קדם עוד
- אי הבדיקה מאפשרת כתיבת מודולים "סובלניים" שיעטפו קריאות למודולים שאינם מניחים דבר על הקלט שלהם
- כך נפריד את בדיקות התקינות מהלוגיקה העסקית (business logic) כלומר ממה שהפונקציה עושה באמת
- גישת תיכון ע"פ חוזה סותרת גישה בשם "תכנות מתגונן" (defensive programming) שעיקריה לבדוק תמיד הכל



חלוקת אחריות

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

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

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

דוגמא

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

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

חיזוק תנאי האחר

- אם תנאי הקדם לא מתקיים, לשירות מותר שלא לקיים את תנאי האחר כשהוא מסיים; קריאה לשירות כאשר תנאי הקדם שלו לא מתקיים מהווה תקלה שמעידה על פגם בתוכנית

- אבל גם אם תנאי הקדם לא מתקיים, **מותר** לשירות לפעול ולקיים את תנאי האחר

- לשירות מותר גם לייצר, כאשר הוא מסיים, מצב הרבה יותר ספציפי מזה המתואר בתנאי האחר; תנאי האחר לא חייב לתאר **בדיוק** את המצב שייווצר אלא מצב כללי יותר (תנאי חלש יותר)
  - למשל, שירות המתחייב לביצוע חישוב בדיוק של ε כלשהו יכול בפועל להחזיר חישוב בדיוק של 2/ ε

דע מה אתה מבקש

- מי מונע מאיתנו לעשות שטויות?
- אף אחד

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

החוזה והקומפיילר

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

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

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

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