תוכנה 1 בשפת Java

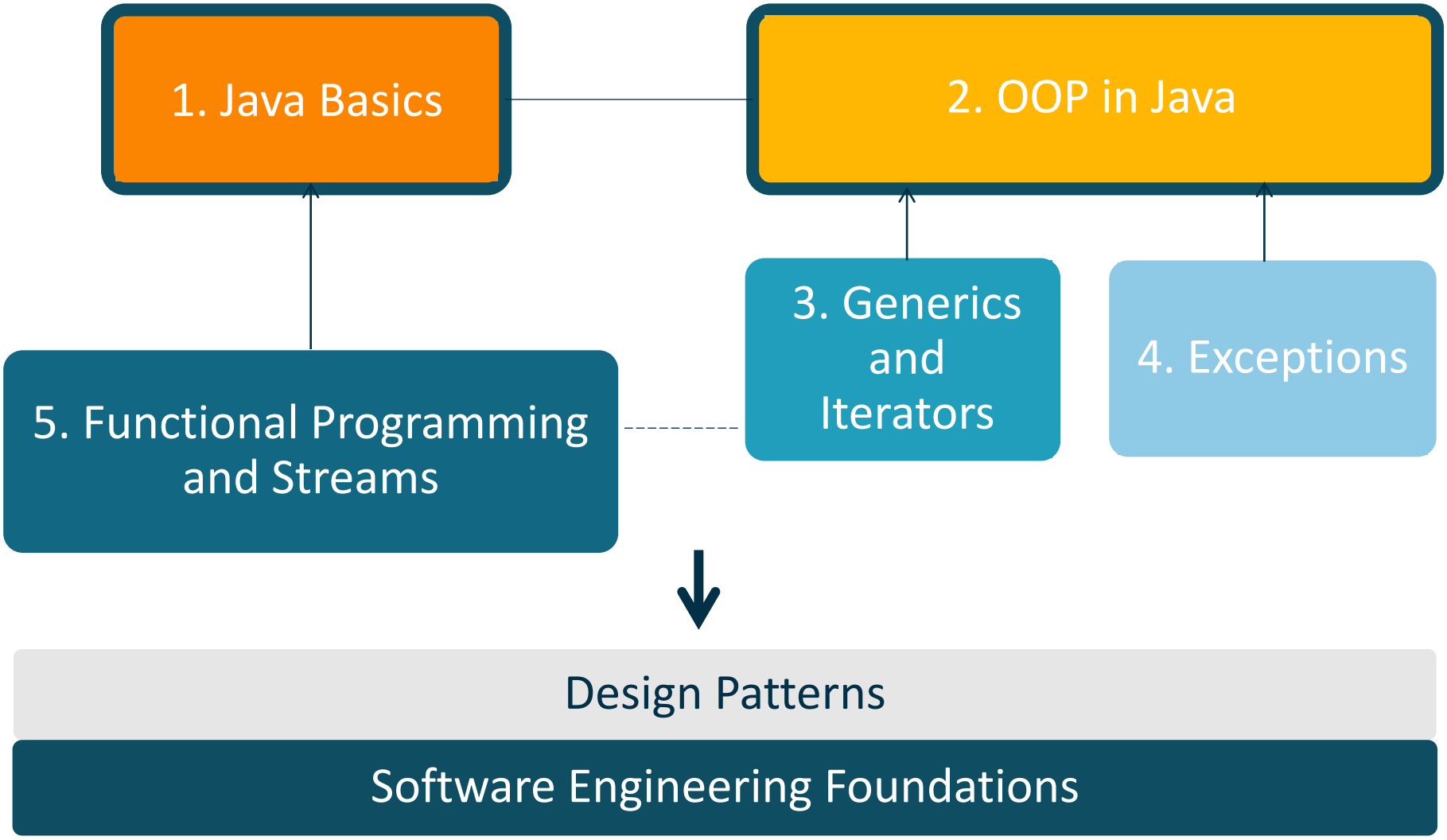
שיעור מספר 4: מה ראינו עד כה? (שבוע השלמת פערים)

מיכל קליינבורט

בית הספר למדעי המחשב

אוניברסיטת תל אביב

נושאי הקורס


1. Java Basics
2. OOP in Java
3. Generics and Iterators
4. Exceptions
5. Functional Programming and Streams
Design Patterns
Software Engineering Foundations

מה ראינו עד כה?

- תכנות ב Java - בסיס
- 8 הטיפוסים היסודיים ב Java, טיפוס המחרוזת וטיפוס המערך
- תחביר Java: פקודות תנאי, לולאות, אופרטורים,switch
- שרותי מחלקה ומשתני מחלקה
- העמסה
- מודל הזיכרון של Java (המחסנית והערימה)
- מחלקות ושירותי מופע
- enum
- חוזים

מה ראינו עד כה?

תכנות ב Java - בסיס

8 הטיפוסים היסודיים ב Java, טיפוס המחרוזת וטיפוס המערך

תחביר Java: פקודות תנאי, לולאות, אופרטורים,switch

שרותי מחלקה ומשתני מחלקה

העמסה

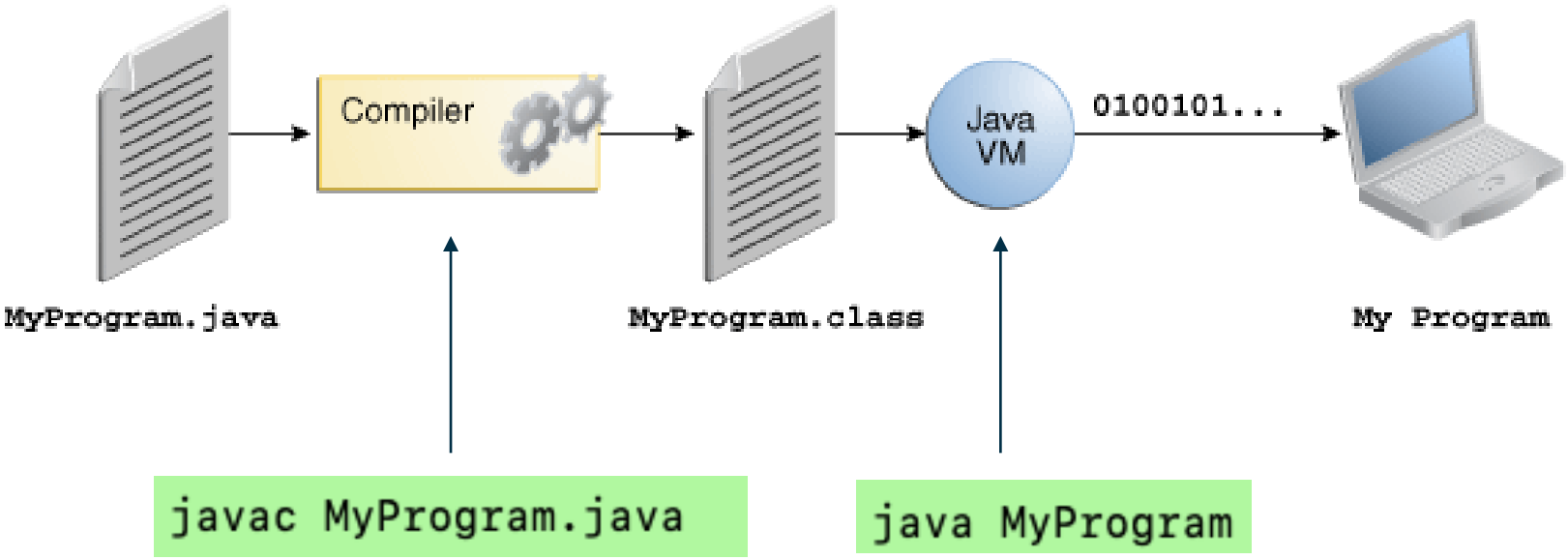
מודל הזיכרון של Java (המחסנית והערימה)

מחלקות ושירותי מופע

enum

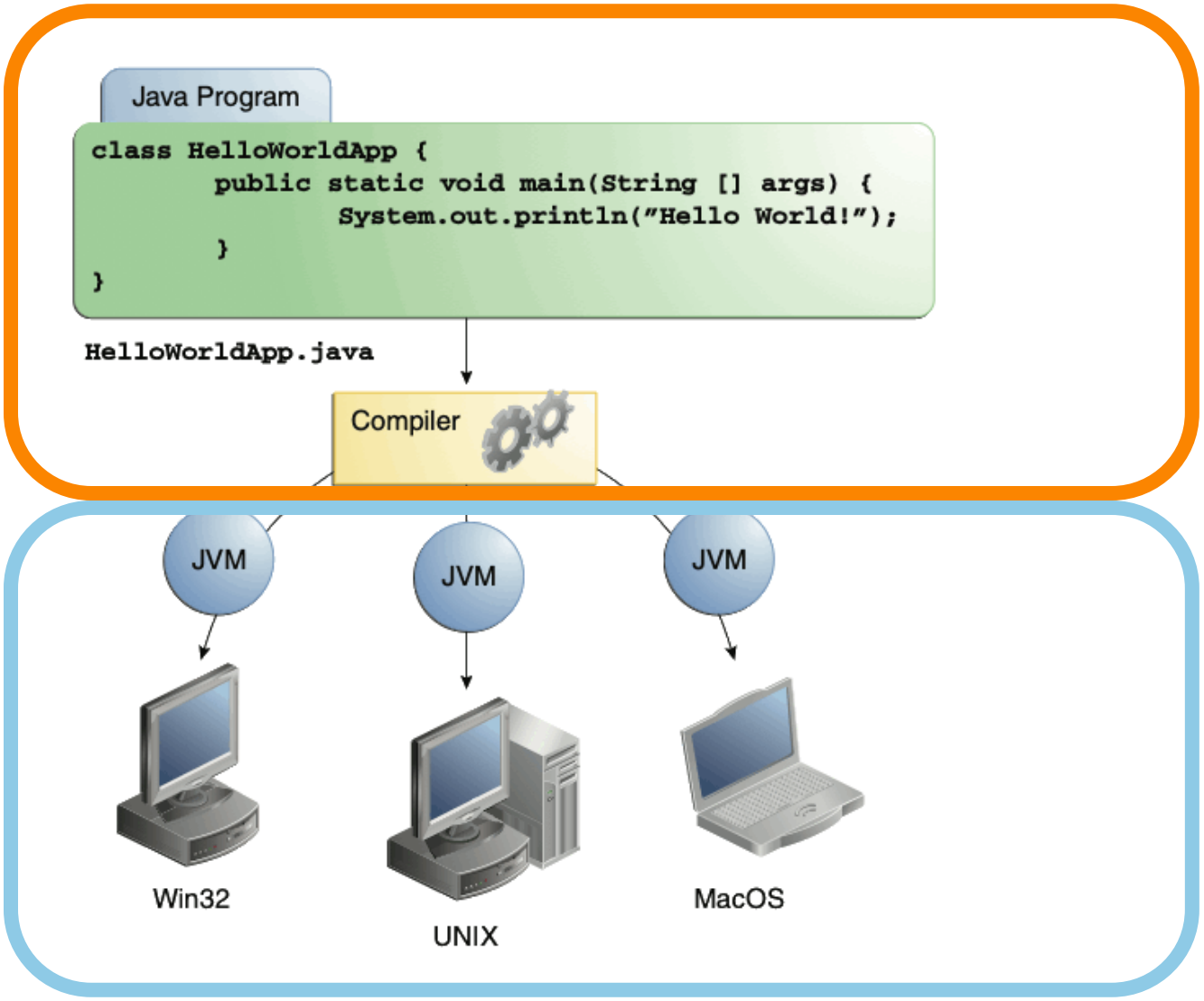
חוזים

שלום עולם



https://docs.oracle.com/javase/tutorial/getStarted/intro/definition.html

שלום עולם



שלב הפיתוח

שלב הריצה

המכונה המדומה (**J**ava **V**irtual **M**achine- **JVM**)

- הקובץ המהודר מכיל הוראות ריצה ב"מחשב כללי" – הוא אינו עושה הנחות על ארכיטקטורת המעבד, מערכת ההפעלה, הזיכרון וכו'...

- עבור כל סביבה (פלטפורמה) נכתב מפרש מיוחד שיודע לבצע את התרגום מהמחשב הכללי, המדומה, למחשב המסוים שעליו מתבצעת הריצה

- את המפרש לא כותב המתכנת!
  - דבר זה כבר נעשה ע"י ספקי תוכנה שזה תפקידם, עבור רוב סביבות הריצה הנפוצות

מה ראינו עד כה?

- תכנות ב Java - בסיס
- 8 הטיפוסים היסודיים ב Java, טיפוס המחרוזת וטיפוס המערך
- תחביר Java: פקודות תנאי, לולאות, אופרטורים,switch
- שרותי מחלקה ומשתני מחלקה
- העמסה
- מודל הזיכרון של Java (המחסנית והערימה)
- מחלקות ושירותי מופע
- enum
- חוזים

טיפוסים בשפת Java

בג'אווה יש שתי משפחות של טיפוסים, ובהתאם לכך שני סוגי משתנים:

- הטיפוסים היסודיים – 8 טיפוסים שהם חלק משפת התכנות, והם מיועדים להכיל ערכים פשוטים (כגון מספרים)
- טיפוסי הפנייה – המייצגים ישויות מורכבות יותר הנקראות מחלקות (כגון מחרוזות, מערכים, קבצים ועוד...). טיפוסים אלו יכולים גם להכיל מידע וגם לספק שרותים

9

הטיפוסים היסודיים (primitive types)

- בג'אווה 8 טיפוסים יסודיים:
  - מספרים שלמים: **`byte, short, int, long`**
  - מספרים בייצוג נקודה צפה: **`float, double`**
  - תווים: **`char`**
  - ערכים בולאנים: **`boolean`**

10

הטיפוסים היסודיים

Type	Contains	Default	Size	Range
boolean	true / false	false	1 bit	
char	Unicode character	\u0000	16 bits	\u0000 to \uFFFF
byte	Signed integer	0	8 bits	-128 to 127
short	Signed integer	0	16 bits	-32768 to 32767
int	Signed integer	0	32 bits	-2147483648 to 2147483647
long	Signed integer	0	64 bits	-9223372036854775808 to 9223372036854775807
float	IEEE 754 floating point	0.0	32 bits	1.4E-45 to 3.4028235E+38
double	IEEE 754 floating point	0.0	64 bits	4.9E-324 to 1.7976931348623157E+308

משתנה מקומי מטיפוס int

```
public static void main(String[] args) {
  int i;
  System.out.println("i=" + i);
  i = 5;
  System.out.println("i=" + i);
```

סימן ה- '+' :
שרשור מחרוזות

- משפט הצהרה - בזיכרון התוכנית (באיזור שנקרא Stack, "המחסנית") מוקצים 4 בתים לצורך שמירת המידע שיוכנס לתוך i
- בנקודת זמן זו, הערך המופיע שם חסר משמעות ("זבל")
- אם נרצה לגשת לנתון כעת זוהי טעות קומפילציה
- זהו משפט השמה – הערך 5 ייכתב לתוך הזיכרון שהוקצה למשתנה i
- כעת, הגישה למשתנה i תקינה ויודפס למסך: i=5
- אין ב Java אתחול ברירת מחדל למשתנים מקומיים

המרת טיפוסים (casting)

- מה קורה אם מנסים להשים לתוך משתנה מטיפוס מסוים ערך מטיפוס אחר?

- תלוי במקרה:
  - אם ההמרה בטוחה (לא יתכן איבוד מידע) – היא בדרך כלל תצליח ללא שגיאות קומפילציה
  - המרה בטוחה של טיפוס נקראת *הרחבה* (widening)

☑
```
int i = 14;
long l = i;
```

- בטיחות ההמרה לא מתייחסת לערך הקיים בפועל, אלא רק לטיפוסו

המרה מפורשת (explicit casting)

- אם ההמרה לא בטוחה?
- בדרך כלל זוהי שגיאת קומפילציה ונדרשת המרה מפורשת:

```
double d = 3.0;
float f = d;
```

- המרה מפורשת היא בקשה מהמהדר לבצע את ההמרה "בכוח", תוך לקיחת אחריות של המתכנת לאיבוד מידע אפשרי
- המרה כזו מכונה הצרה (narrowing)
- המרה מפורשת מתבצעת ע"י ציון הטיפוס החדש בסוגריים לפני הערך שאותו מבקשים להמיר:

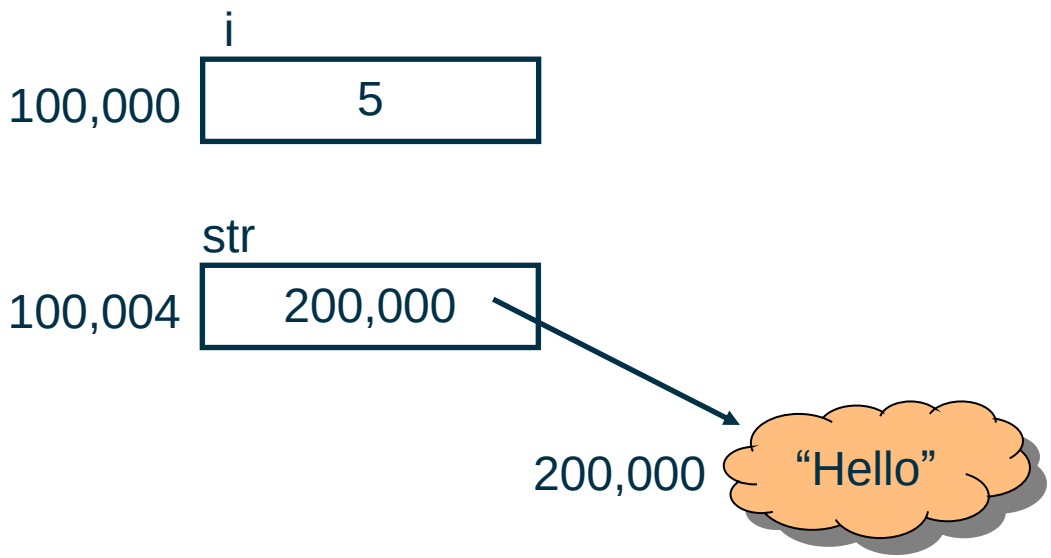
```
double d = 3.0;
float f = (float)d;
```

הפניות ומשתנים יסודיים

- ביצירת משתנה מטיפוס יסודי אנו יוצרים מקום בזיכרון בגודל ידוע שיכול להכיל ערך מטיפוס מסוים
- ביצירת משתנה הפנייה אנו יוצרים מקום בזיכרון, שיכול להכיל כתובת של מקום אחר בזיכרון שם נמצא תוכן כלשהו

➡ int i = 5;

➡ String str = “Hello”



הפניות ועצמים

- המשתנה str נקרא *הפנייה*, התוכן שעליו הוא מצביע נקרא *עצם* (object)

- אזור הזיכרון שבו נוצרים עצמים שונה מאזור הזיכרון שבו נוצרים משתנים מקומיים והוא מכונה Heap (זיכרון ערימה)
  - תזכורת: משתנים מקומיים נוצרים באזור שנקרא Stack ("המחסנית")

- למה חץ? ⟶
  - מכיוון ש Java לא מרשה למתכנת לראות את התוכן של משתנה מטיפוס הפנייה (בשונה משפת C)

- למה ענן? 
  - מכיוון שאנו לא יודעים את מבנה הזיכרון שבו מיוצגים טיפוסים שאינם יסודיים

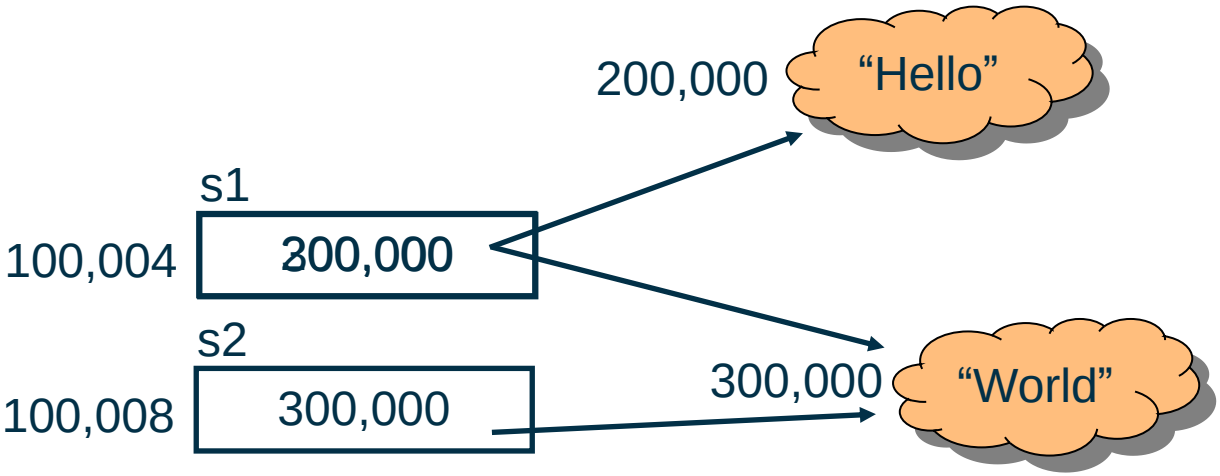
פעולות על הפניות

- השמה למשתנה הפנייה שמה ערך חדש במשתנה ההפנייה ללא קשר לעצם המוצבע!

➡ String s1 = “Hello”;

➡ String s2 = “World”;

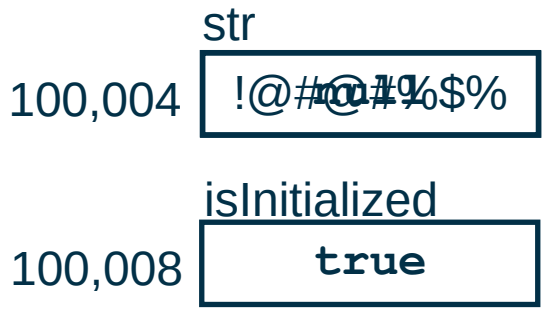
➡ s1= s2;



ערך null

- ניתן לייצר משתנה הפנייה ללא אתחולו. כמו ביצירת משתנה פרימיטיבי ערכו יהיה זבל, ולא ניתן יהיה לגשת אליו
- ניתן להשים למשתנה הפנייה את הערך null (לא מוגדר). כך ניתן יהיה לגשת אליו בהמשך כדי לבדוק אם אותחל

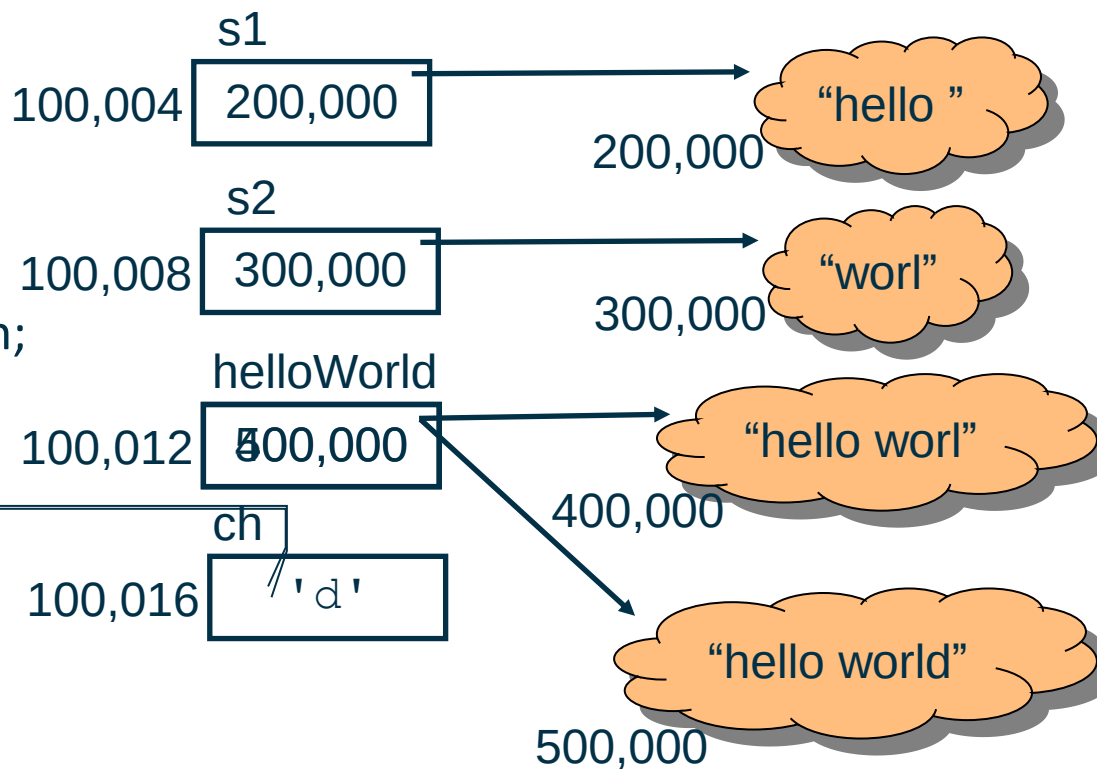
```
String str;
str = null;
boolean isInitialized = (str == null);
```



שרשור מחרוזות

- באשר אחד האופרנדים של אופרטור ה '+' הוא מחרוזת, הוא מתרגם את כל שאר האופרנדים למחרוזת ומייצר מחרוזת חדשה שהיא שרשור כל המחרוזות

- String s1 = "hello ";
- String s2 = "worl";
- String helloWorld = s1 + s2;
- char ch = 'd';
- helloWorld = helloWorld + ch;



מערכים

- מייצג סדרת משתנים מאותו טיפוס (בין אם פרימיטיבי או הפניה)
- למשל מערך של איברים מטיפוס int:

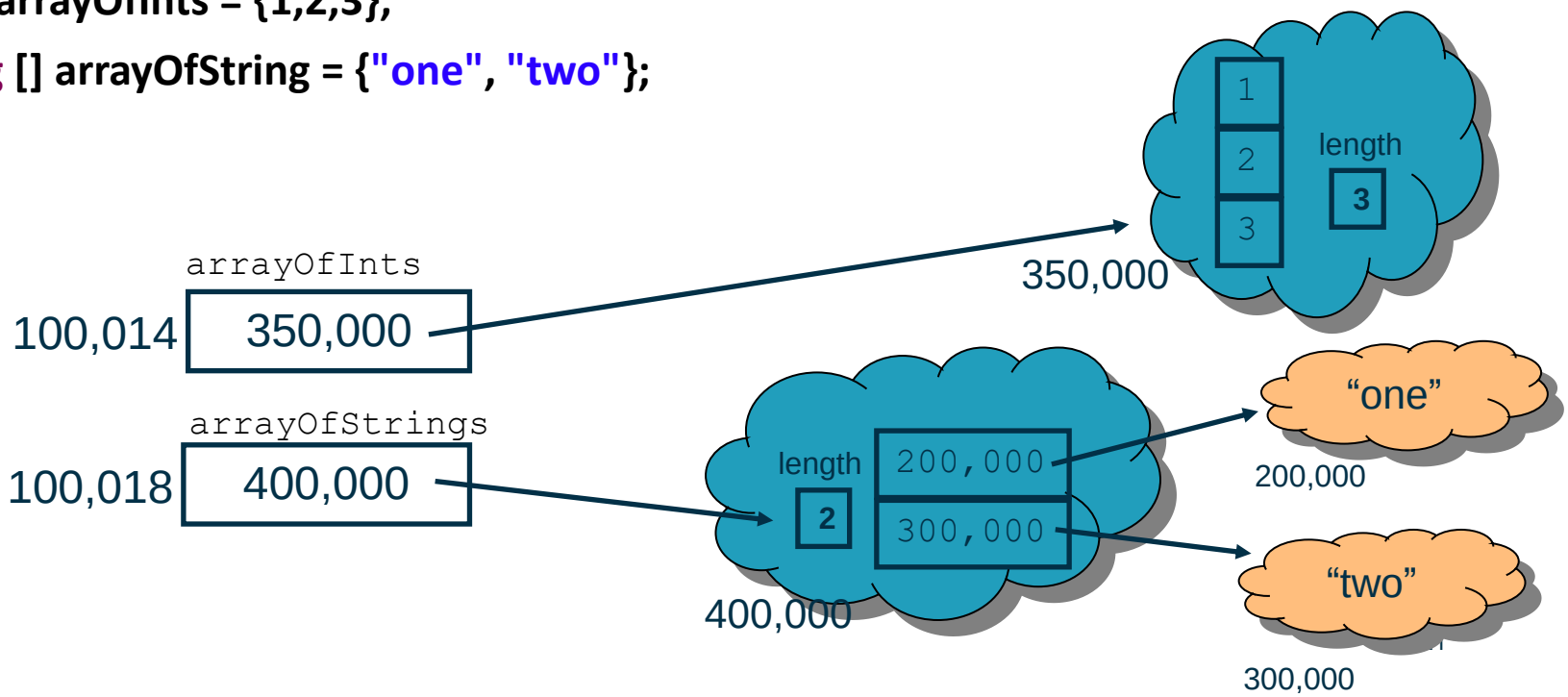
2	3	5	7	11	13	17	19	23	29

- תאים במערך יושבים בדרך כלל ברצף בזיכרון (Java רצה על מכונה וירטואלית!) כך שגישה סדרתית אליהם עשויה להיות יעילה
- מערך אינו זהה לטיפוס list שאתם מכירים מ Python!

מערכים

- גם מערכים אינם חלק מהטיפוסים היסודיים של Java ועל כן משתנה מערך הוא מטיפוס הפנייה
- בדי לציין שמשתנה הוא מטיפוס מערך נשתמש בסוגריים המרובעים ("מרובעיים")

```
int [] arrayOfInts = {1,2,3};
String [] arrayOfString = {"one", "two"};
```



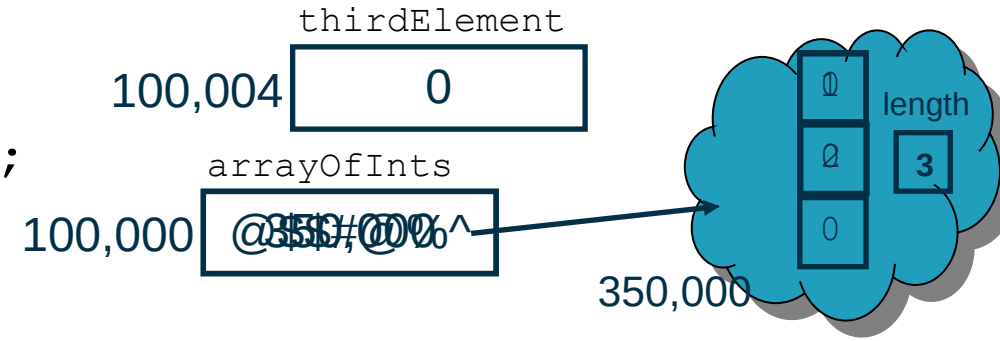
מערכים

- נשים לב להבדל בין מערך של טיפוס פרימיטיבי ומערך של טיפוס הפנייה:
  - במערכים של טיפוסים פרימיטיביים, הערכים הפרימיטיביים יושבים במערך עצמו (במקום שהוקצה לו בזכרון)
  - במערכים של טיפוס הפנייה, הערכים הנמצאים במערך הן הפניות לעצמים הנמצאים במקום אחר בזכרון

- בשקף הקודם ראינו *אתחול* של מערך בעזרת שימוש בסוגריים מסולסלים. אם נרצה להפריד בין יצירת ההפנייה ואתחולה (יצירת עצם המערך) יש להשתמש באופרטור **new**
- כדי לגשת לאיבר מסוים במערך (קריאה או כתיבה) נשתמש באופרטור הסוגריים המרובעים

יצירת עצם מטיפוס מערך וגישה לאיבריו

```
int [] arrayOfInts;
arrayOfInts = new int[3];
arrayOfInts[0] = 1;
arrayOfInts[1] = 2;
int thirdElement = arrayOfInts[2];
```



איברי מערך שהוקצה ע"י **new** מאותחלים אוטומטית לפי טיפוסם:

- הטיפוסים הפרימיטיביים השלמים מאותחלים ל-0
- הטיפוסים הפרימיטיביים הממשיים מאותחלים ל- 0.0
- הטיפוס הפרימיטיבי **boolean** מאותחל ל- **false**
- הטיפוס הפרימיטיבי **char** מאותחל לתו שערך ה Unicode שלו הוא 0
- טיפוס הפנייה מאותחל ל- **null**

ניתן לשאול מערך לאורכו

- אורכו של מערך, הוא מאפיין פנימי אשר ניתן לגשת אליו ישירות בעזרת אופרטור הנקודה

```
int [] arrayOfInts = {1,2,3};
System.out.println("The size of my array is " +
 arrayOfInts.length);
```

הפרוטקציה של מערכים ומחרוזות

- מכיוון שמחרוזות ומערכים הם טיפוסים מאוד שכיחים ושימושיים בשפה, הם קיבלו "יחס מועדף", שתי תכונות שאין לאף טיפוס אחר בשפה:

- **פטור מ- new**
  - לא ניתן ב Java לייצר עצם ללא שימוש מפורש באופרטור **`new`**

  אבל
  - ניתן ליצור עצם מחרוזת ע"י שימוש בסימן המרכאות ("**`hello`**"), ניתן ליצור עצם מערך ע"י שימוש במסולסליים (**{1,2,3}**)

- **הפניות ואופרטורים**
  - על משתנה מטיפוס הפניה אפשר לבצע רק השמה (אופרטור '='), השוואה (אופרטור '==') או גישה לעצם (אופרטור '.')

  אבל
  - על מערך ניתן גם לבצע גישה לאיבר ([ ]), על מחרוזת ניתן לבצע גם שרשור (+)

מה ראינו עד כה?

- תכנות ב Java - בסיס
- 8 הטיפוסים היסודיים ב Java, טיפוס המחרוזת וטיפוס המערך
- תחביר Java: פקודות תנאי, לולאות, אופרטורים ,switch
- שרותי מחלקה ומשתני מחלקה
- העמסה
- מודל הזיכרון של Java (המחסנית והערימה)
- מחלקות ושירותי מופע
- enum
- חוזים

מה ראינו עד כה?

- תכנות ב Java - בסיס
- 8 הטיפוסים היסודיים ב Java, טיפוס המחרוזת וטיפוס המערך
- תחביר Java: פקודות תנאי, לולאות, אופרטורים ,switch
- **שרותי מחלקה ומשתני מחלקה**
- העמסה
- מודל הזיכרון של Java (המחסנית והערימה)
- מחלקות ושירותי מופע
- enum
- חוזים

שרותי מחלקה (static methods)

- *שרות מחלקה* הוא סדרת משפטים שניתן להפעילה ממקום אחר בקוד ע"י קריאה לשרות, והעברת אפס או יותר ארגומנטים

- שירותים כאלה מובחנים על ידי מילת המפתח **`static`** כמו למשל:

```
public class MethodExamples {

  public static void printMultipleTimes(String text, int times) {
     for(int i=0; i<times; i++)
        System.out.println(text);
  }

}
```

- נתעלם כרגע מההכרזה **`public`**

הגדרת שרות

- התחביר של הגדרת שרות הוא:

```
<modifiers> <type> <method-name> ( <paramlist> ) {
  <statements>
}
```

- <modifiers> הם 0 או יותר מילות מפתח מופרדות ברווחים (למשל public static)
- <type> מציין את טיפוס הערך שהשרות מחזיר
  - **void** מציין שהשרות אינו מחזיר ערך
- <paramlist> רשימת הפרמטרים הפורמליים, מופרדים בפסיק, כל אחד מורכב **מטיפוס** הפרמטר **ושמו**
- **חתימה של שרות** כוללת את שם השרות והפרמטרים הפורמלים שלו

גוף השרות

- גוף השרות מכיל הצהרות על משתנים מקומיים (variable declaration)
- משתנים מקומיים נקראים גם משתנים זמניים, משתני מחסנית או משתנים אוטומטיים

```
public static void doSomething(String str) {
  int length = str.length();
  ...
```

- הגדרת משתנה זמני צריכה להקדים את השימוש בו
- תחום הקיום של המשתנה הוא גוף השרות
- חייבים לאתחל או לשים ערך באופן מפורש במשתנה לפני השימוש בו

שם מלא (qualified name)

- אם אנו רוצים לקרוא לשרות מתוך שרות של מחלקה אחרת (למשל main), יש להשתמש בשמו המלא של השרות

- שם מלא כולל את שם המחלקה שבה הוגדר השרות ואחריו נקודה

```
public class CallingFromAnotherClass {

    public static void main(String[] args) {
      printMultipleTimes("Hello", 5);

        int[] primes = { 2, 3, 5, 7, 11 };
        System.out.println("Sum of primes is: " +
      arraySum(primes));
    }
}
```

שם מלא (qualified name)

- במחלקה המגדירה ניתן להשתמש בשם המלא של השרות, או במזהה הפונקציה בלבד (unqualified name)
- בצורה זו ניתן להגדיר במחלקות שונות פונקציות עם אותו השם (מכיוון שהשם המלא שלהן שונה אין התלבטות – no ambiguity)

```
public class CallingFromAnotherClass {

    public static void main(String[] args) {
        MethodCallExamples.printMultipleTimes("Hello", 5);

        int[] primes = { 2, 3, 5, 7, 11 };
        System.out.println("Sum of primes is: " +
                           MethodCallExamples.arraySum(primes));
        }
}
```

- כבר ראינו **שימוש אחר** באופרטור הנקודה כדי לבקש בקשה מעצם, זהו שימוש נפרד שאינו שייך להקשר זה

משתני מחלקה

- עד כה ראינו משתנים מקומיים – משתנים זמניים המוגדרים בתוך מתודה, בכל קריאה למתודה הם נוצרים וביציאה ממנה הם נהרסים

- ב Java ניתן גם להגדיר **משתנים גלובליים** (global variables)
  - מכונים שדות סטטיים (static fields/members)

- משתנים אלו יוגדרו בתוך גוף המחלקה אך מחוץ לגוף של מתודה כלשהי, ויסומנו ע"י המציין `static`.
  - יכולה להיות ניראות שונה (public/private)

משתני מחלקה לעומת משתנים מקומיים

- משתנים אלו, שונים ממשתנים מקומיים בכמה מאפיינים:
  - תחום הכרות: כתלות בנראות (נראה נראויות שונות בהמשך הקורס), מוברים בכל הקוד, ולא רק בתוך פונקציה מסויימת
  - משך קיום: אותו עותק של משתנה נוצר בזמן טעינת הקוד לזיכרון ונשאר קיים בזיכרון התוכנית כל עוד המחלקה בשימוש
  - אתחול: משתנים סטטיים מאותחלים בעת יצירתם. אם המתכנתת לא הגדירה להם ערך אתחול - יאותחלו לערך ברירת המחדל לפי טיפוסם (`0, false, null`)
  - הקצאת זיכרון: הזיכרון המוקצה להם נמצא באזור ה Heap (ולא באזור ה-Stack)

דוגמה: שימוש במשתנה גלובלי **counter** בדי לספור את מספר הקריאות למתודה **m()**

```
public class StaticMemberExample {
  public static int counter; //initialized by default to 0;

  public static void m() {
     int local = 0;
     counter++;
     local++;
     System.out.println("m(): local is " + local +
                        "\tcounter is " + counter);
  }

  public static void main(String[] args) {
     m();
     m();
     m();
     System.out.println("main(): m() was called " +
                                    counter + " times");
  }
}
```

שם מלא

- ניתן לפנות למשתנה counter גם מתוך קוד במחלקה אחרת, אולם יש צורך לציין את שמו המלא (qualified name)
- במחלקה שבה הוגדר משתנה גלובלי ניתן לגשת אליו תוך ציון שמו המלא או שם המזהה בלבד (unqualified name)
- בדומה לצורת הקריאה לשרותי מחלקה

```
public class AnotherClass {

    public static void main(String[] args) {
        StaticMemberExample.m();
        StaticMemberExample.m();
        StaticMemberExample.m();
        System.out.println("main(): m() was called " +
           StaticMemberExample.counter + " times");
        }
}
```

זה סופי

- ניתן לקבע ערך של משתנה (משתנה מקומי או משתנה מחלקה) ע"י ציון המשתנה כ **`final`**
- למשתנה שהוא **`final`** ניתן לבצע השמה פעם אחת בדיוק. כל השמה נוספת לאותו משתנה תגרור שגיאת קומפילציה
- דוגמה:

```
public final static long uniqueID = ++counter;
```

- מוסכמה מקובלת היא להגדיר שמות משתנים המציינים קבועים ב-UPPERCASE, כגון:

```
public final static double FOOT = 0.3048;
public final static double PI = 3.14159265358979323384;
```

מה ראינו עד כה?

- תכנות ב Java - בסיס
- 8 הטיפוסים היסודיים ב Java, טיפוס המחרוזת וטיפוס המערך
- תחביר Java: פקודות תנאי, לולאות, אופרטורים ,switch
- שרותי מחלקה ומשתני מחלקה
- העמסה
- מודל הזיכרון של Java (המחסנית והערימה)
- מחלקות ושירותי מופע
- enum
- חוזים

העמסת שרותים (method overloading)

- לשתי פונקציות ב Java יכול להיות אותו שם (מזהה) גם אם הן באותה מחלקה, ובתנאי שהחתימה שלהן שונה
  - כלומר הן שונות בטיפוס ו/או במספר הארגומנטים שלהם
  - לא כולל ערך מוחזר!

- הגדרת שתי פונקציות באותו שם ובאותה מחלקה נקראת **העמסה**

- ראינו שלוש סיבות לשימוש בתכונת ההעמסה
  - נוחות
  - ערכי ברירת מחדל לארגומנטים
  - תאימות אחורה

העמסה והקומפיילר

- המהדר מנסה למצוא את הגרסה המתאימה ביותר עבור כל קריאה לפונקציה על פי טיפוסי הארגומנטים של הקריאה
- אם אין התאמה מדויקת לאף אחת מחתימות השרותים הקיימים, המהדר מנסה המרות (casting) ש(כמעט)-אינן מאבדות מידע.
- ראו פרק 5, **Conversions and Contexts** בקישור:

  https://docs.oracle.com/javase/specs/jls/se21/html/jls-5.html
- אם לא נמצאת התאמה או שנמצאות שתי התאמות "באותה רמת סבירות" או שפרמטרים שונים מתאימים לפונקציות שונות המהדר מודיע על אי בהירות (ambiguity)

העמסה, שכפול קוד ועקביות

- חסרונות שכפול קוד:

  קוד שמופיע פעמיים, יש לתחזק פעמיים – כל שינוי, שדרוג או תיקון עתידי יש לבצע בצורה עקבית בכל המקומות שבהם מופיע אותו קטע הקוד

- בדי לשמור על עקביות שתי הגרסאות של **compute** נממש את הגרסה הישנה בעזרת הגרסה החדשה:

```
public static int compute(int x, int base){
  // complex calcultion...
}

public static int compute(int x){
   return compute(x, 10);
}
```

שכפול קוד הוא הדבר הנורא ביותר בעולם (התוכנה)!

מה ראינו עד כה?

- תכנות ב Java - בסיס
- 8 הטיפוסים היסודיים ב Java, טיפוס המחרוזת וטיפוס המערך
- תחביר Java: פקודות תנאי, לולאות, אופרטורים ,switch
- שרותי מחלקה ומשתני מחלקה
- העמסה
- **מודל הזיכרון של Java (המחסנית והערימה)**
- מחלקות ושירותי מופע
- enum
- חוזים

העברת ארגומנטים

- כאשר מתבצעת קריאה לשרות, ערכי הארגומנטים נקשרים לפרמטרים הפורמליים של השרות לפי הסדר, ומתבצעת השמה לפני ביצוע גוף השרות.

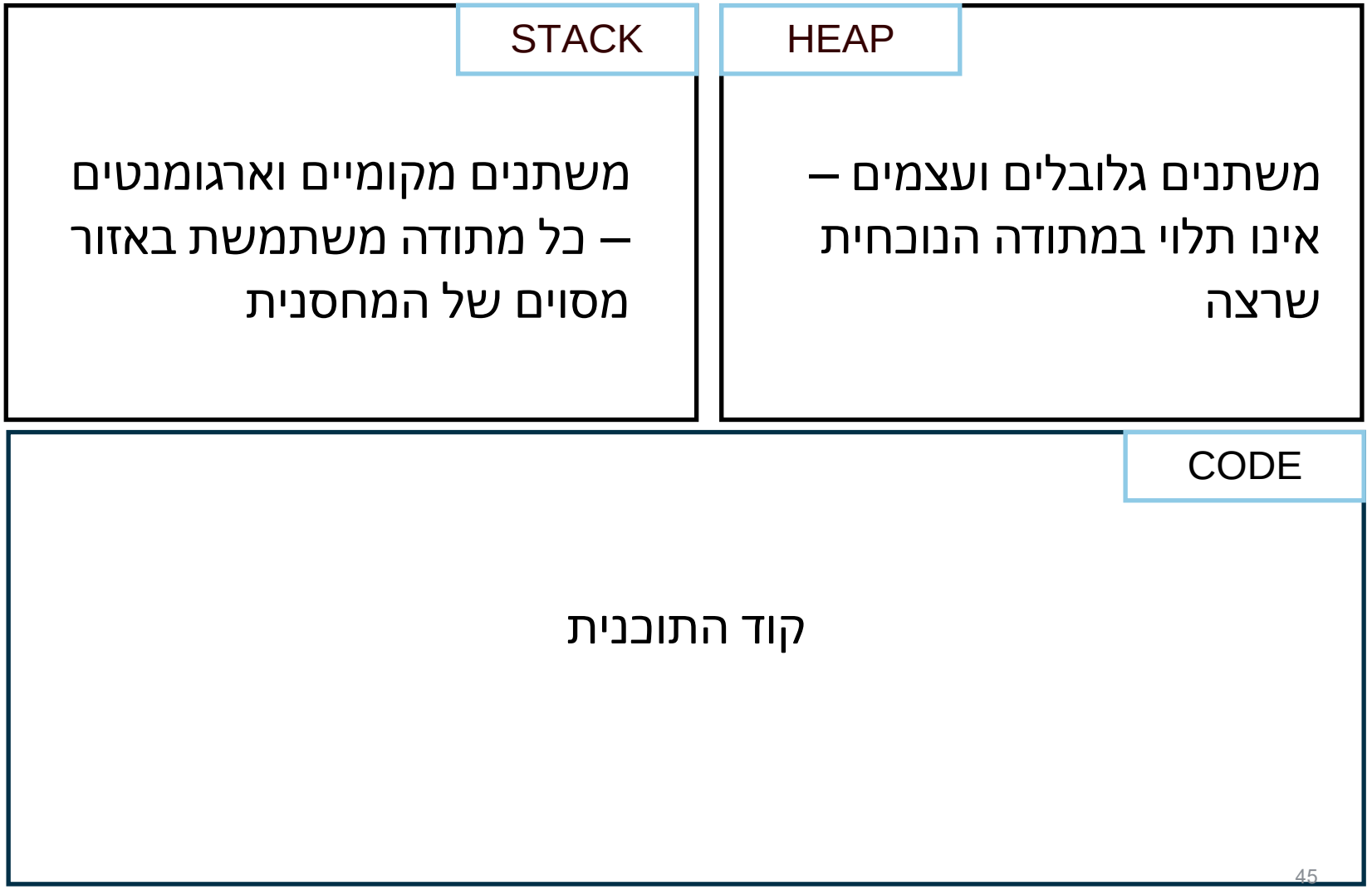
- בהעברת ערך לשרות הערך **מועתק** לפרמטר הפורמלי

- צורה זאת של העברת פרמטרים נקראת **call by value**

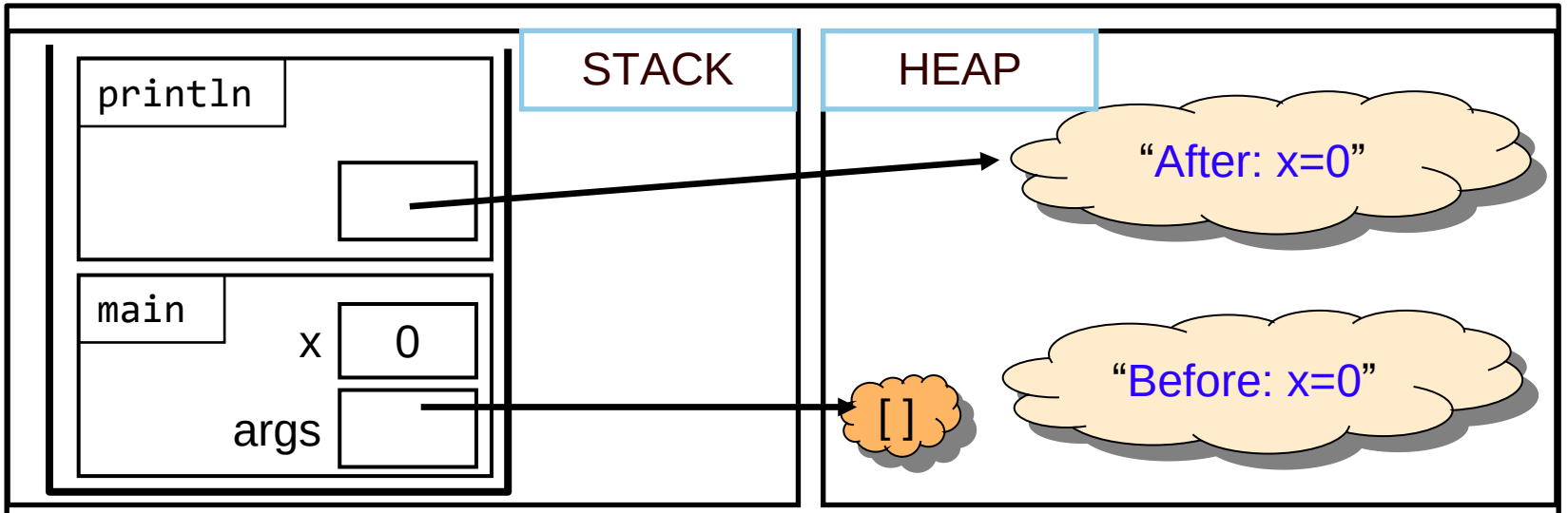
- כאשר הארגומנט המועבר הוא **הפנייה** (התייחסות, reference) העברת הפרמטר **מעתיקה את ההתייחסות.**
  - בשפות תכנות אחרות ניתן לבצע העברה של המצביע עצמו

ב Java גם reference מועבר by value

מודל הזיכרון של Java



Primitives by value




```
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
איבוד הקשר לכתובת שהוקצתה עבורה
על ה-Stack. משוחרר

שמות מקומיים

- בדוגמה 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

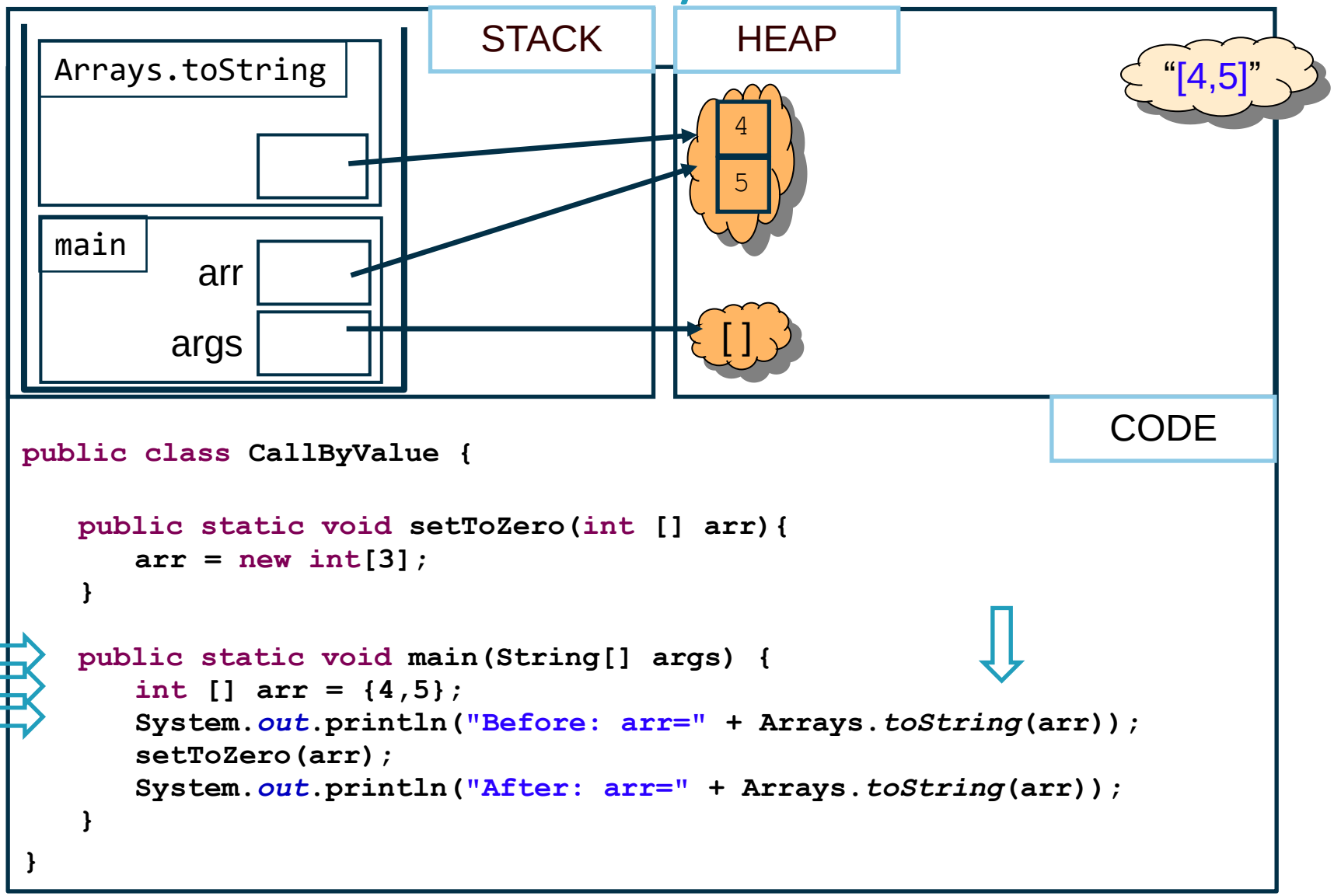
- שם של משתנה מקומי **מסתיר** משתנים בשם זהה הנמצאים בתחום עוטף או גלובליים

- מתודה מבירה רק משתני מחסנית הנמצאים באזור שהוקצה לה על המחסנית (frame)

מה יקרה אם המשתנה המקומי שהועבר היה מטיפוס הפנייה? מה ידפיס הקוד הבא?

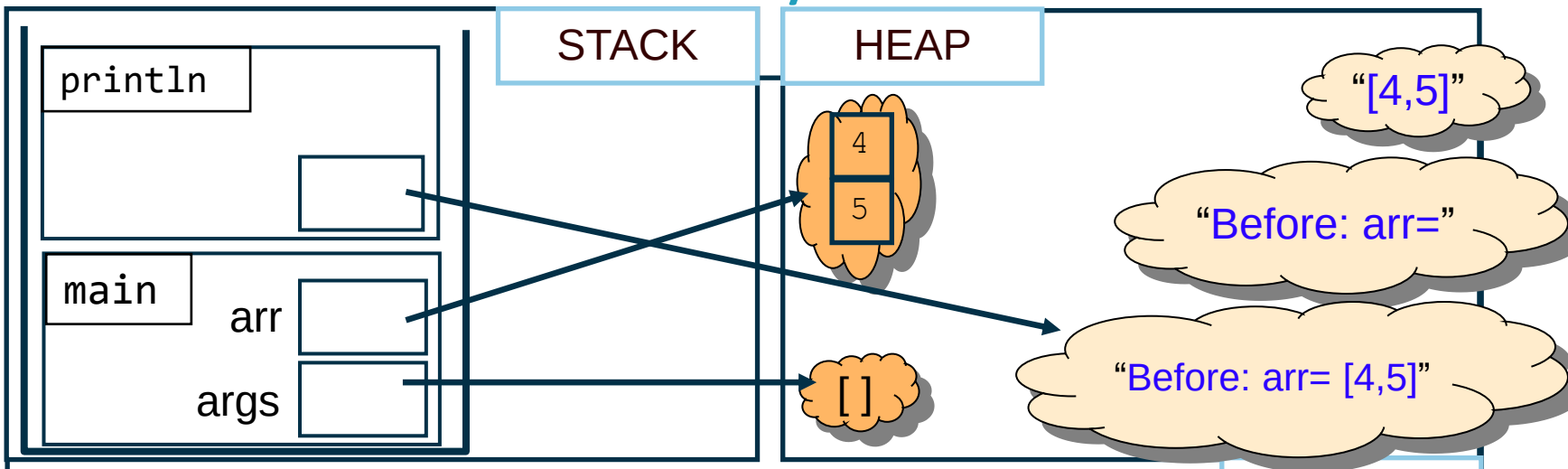
```
import java.util.Arrays; //explained later...

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



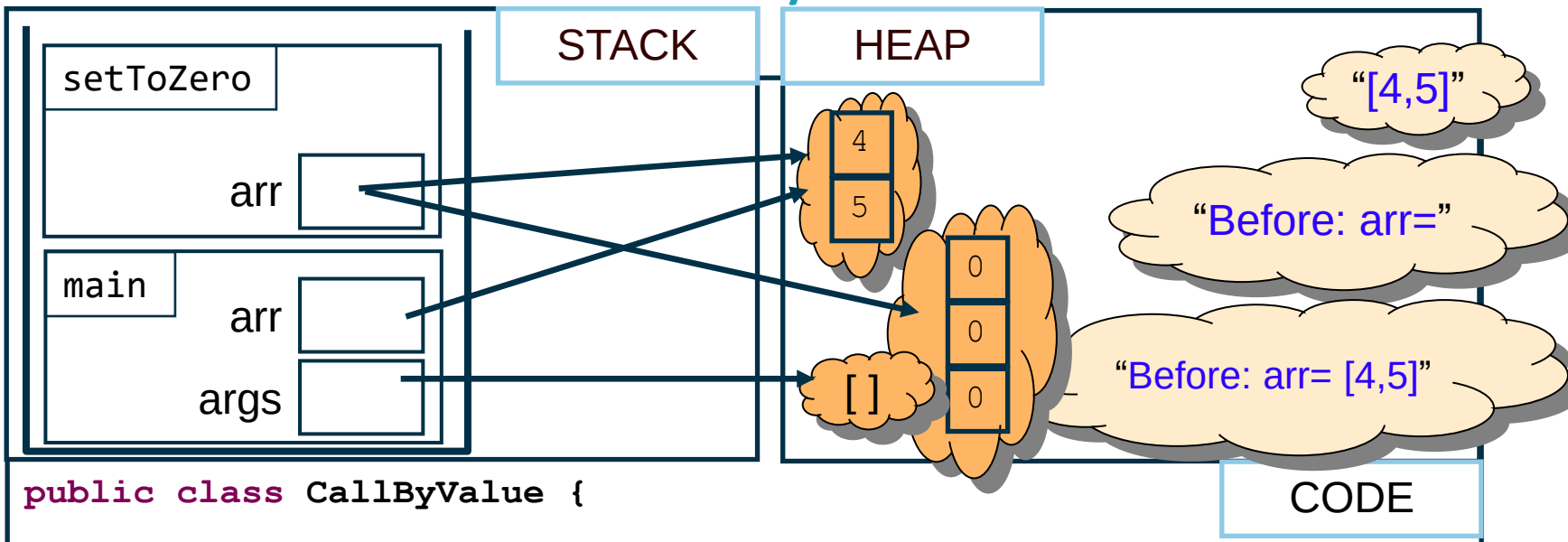

```
public class CallByValue {

    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



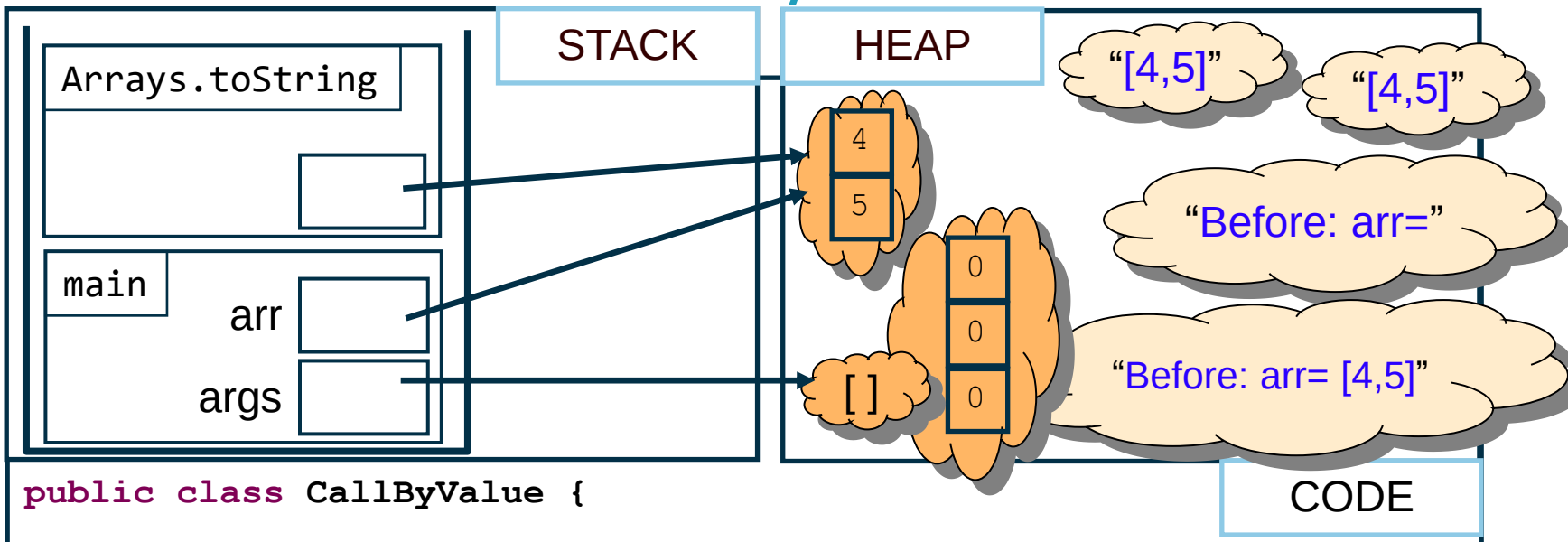

```
public class CallByValue {

    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



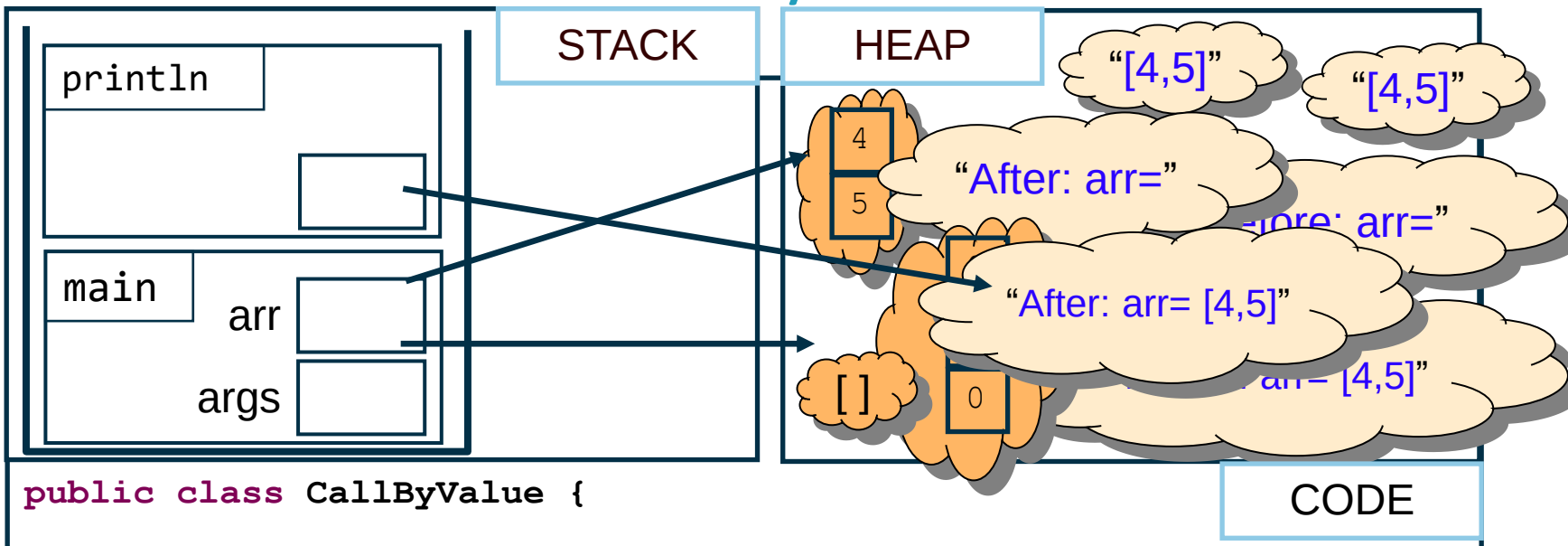

```
public class CallByValue {

    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




```
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

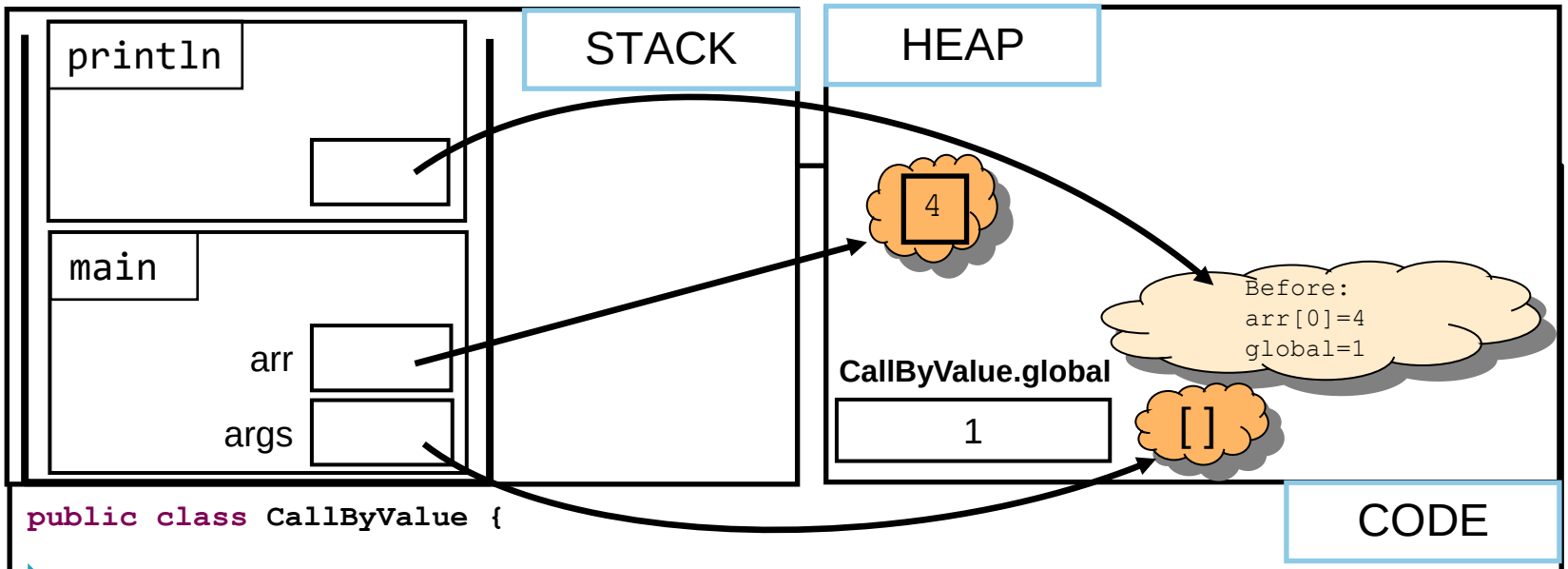
- בשיטת העברה by value לא יעזור למתודה לשנות את הארגומנט שקיבלה, מכיוון שהיא מקבלת עותק

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

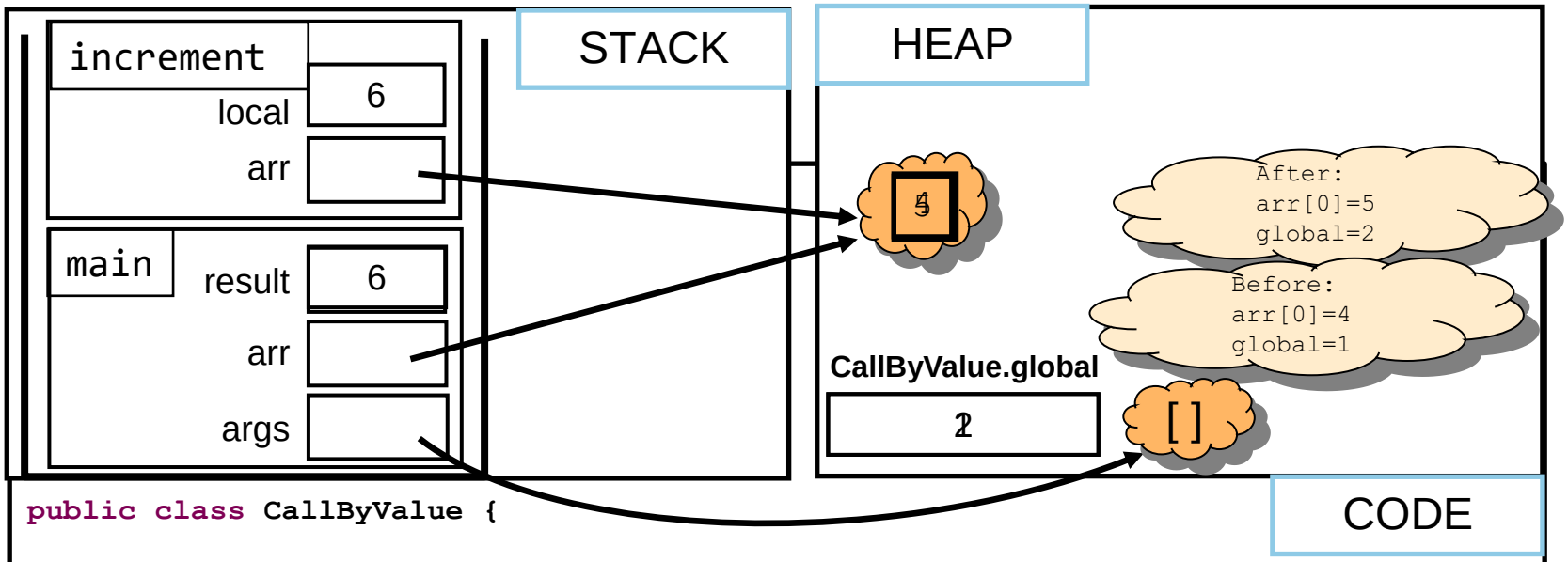
- מתודות שמשנות את תמונת הזיכרון נקראות בהקשרים מסוימים Mutators או Transformers

מה מדפיסה התוכנית הבאה?

```
public class CallByValue {

   static int global = 1;

   public static int increment(int [] arr){
      int local = 6;
      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);
   }
}
```

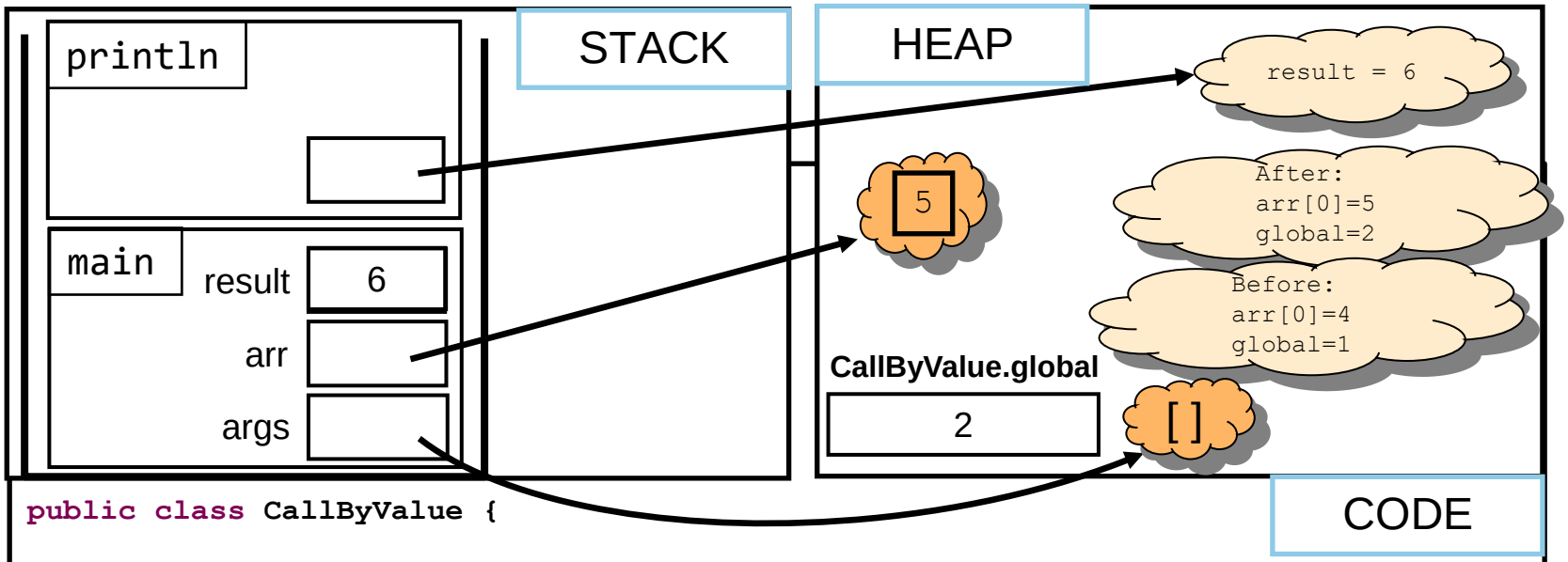



```
public class CallByValue {

    static int global = 1;

    public static int increment(int [] arr){
        int local = 6;
        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 = 1;

     public static int increment(int [] arr){
          int local = 6;
          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!




```
public class CallByValue {

     static int global = 1;

     public static int increment(int [] arr){
          int local = 6;
          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 למיניהם) מיד לאחר טעינת המחלקה לזיכרון, פעולות אלו יש לציין בתוך בלוק **`static`**
  - פרטים נוספים:
    https://docs.oracle.com/javase/tutorial/java/javaOO/initial.html

מה ראינו עד כה?

- תכנות ב Java - בסיס
- 8 הטיפוסים היסודיים ב Java, טיפוס המחרוזת וטיפוס המערך
- תחביר Java: פקודות תנאי, לולאות, אופרטורים ,switch
- שרותי מחלקה ומשתני מחלקה
- העמסה
- מודל הזיכרון של Java (המחסנית והערימה)
- **מחלקות ושירותי מופע**
- enum
- חוזים

מחלקות בטיפוסי 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)

שרותי מופע

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

- תחביר של הפעלת שרות מופע הוא:

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

לדוגמה:

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

- זאת בשונה מזימון שרות מחלקה (static):

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

לדוגמה:

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

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

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?)

דוגמה

- נתבונן 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

- שאלה: כאשר ה JVM טוען לזיכרון את המחלקה איפה בזיכרון נמצאים השדות **day**, **month** ו- **year**?
- תשובה: 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
- דוגמה:

```
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);
    }
}
```

- בדוגמה 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

- המחלקה היא גם מודול. אחריותו של הספק – כותב המחלקה – לממש את כל הלוגיקה הנלווית לייצוג תאריכים
  - בדי לאכוף את עקביות המימוש על משתני המופע להיות פרטיים

שרותי מופע

- שירותי מופע הם שרותים המשויכים למופע מסוים – 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, כלומר – לדמיין את שרות המופע **m()** של מחלקה **C** באילו היה שרות מחלקה (סטטי) המקבל עצם מהטיפוס **C** בארגומנט:

```
public class C {

    public void m(args){ … }

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

m הוא שירות מופע

```
public class C {

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

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

m הוא שירות סטטי

"לא מה שחשבת"

- שרותי מופע מספקים תכונה נוספת ל Java פרט לסובר התחבירי

- בהמשך הקורס נראה כי לשרותי המופע ב Java תפקיד מרכזי **בשיגור שרותים דינאמי** (dynamic dispatch), תכונה בשפה המאפשרת החלפת המימוש בזמן ריצה ופולימורפיזם

- תיאור שרותי מופע כסובר תחבירי הוא פשטני (ושגוי!) אך נותן **אינטואיציה טובה** לגבי פעולת השרות בשלב זה של הקורס

מימוש באמצעות שרותי מחלקה.
זהו מימוש לא מוצלח

```
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...
}
```

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

אולם לא חובה להשתמש בו

```
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...
}
```

```
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...
}
```

נראות פרטית

- מכיוון שהשדות **day**,**month** 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

בנאים (constructors)

- כדי לפתור את הבעיה שהעצם אינו מכיל ערך תקין מיד עם יצירתו נגדיר עבור המחלקה **בנאי**

- בנאי הוא פונקציית אתחול הנקראת ע"י אופרטור ה **`new`** מיד אחרי שהוקצה מקום לעצם החדש. שמה בשם 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* ו*מצב מופשט בעל משמעות*)
  - למשל, נרצה שהאובייקט ה **`MyDate`** שאנחנו מייצרים יכיל תאריך חוקי מיד עם יצירתו

```
public class MyDate {

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

   // ...
}
```

הגדרת בנאי ל MyDate

```
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;
    }

}
```

בעת מופיעה בקוד ההשמה הבאה:

```
day=day;
```

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

בנאים

```
public class MyDate {

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


   public MyDate(int day, int month, int year) {
      this.day = day;
      this.month = month;
      this.year = year;
      this.hour = 11;
   }
}
```

הגדרת שדות

אתחול שדות

לא ניתן לאתחל שדה שלא הוגדר, בדיוק כשם
שלא ניתן לאתחל ערך של משתנה שלא הוגדר!

בנאי ברירת מחדל

- במידה שלא 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 מופע שלא אותחלו בבנאי?

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

- זיכרון 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 רק כאשר לא הוגדר אף בנאי אחר במחלקה.
אם קיים מימוש של בנאי כלשהו, הבנאי הריק לא נוצר אוטומטית ויש לממש אותו בקוד במידה ונרצה להשתמש בו.

בנאים

- האם הקוד הבא יתקמפל?

```
public class MyDate {

    public MyDate() {
        this.day = 1;
        this.month = 1;
        this.year = 1970;
    }

    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();
    }
}
```

בנאים

```
public class MyDate {

    public MyDate() {
        this(1,1,1970);
    }


    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();
    }
}
```

נשתמש ב **this(args)** בשביל לקרוא לבנאי אחר של המחלקה.
שימו לב! הקריאה ל **this** חייבת להופיע בשורת הקוד הראשונה של הבנאי

records

- החל מ- Java 16
- קיים סוג מסוים של מחלקות עבורן קיים תחביר מיוחד
- מחלקות אלה מייצגות "plain data carriers" בלומר, מחלקות שמאגדות יחד במה פרטי מידע שאמורים לעבור ממקום למקום
  - למשל, מזהה של סטודנט שמורכב מתעודת זהות ושם
- מחלקות אלה מיוצרות עם מידע בלשהו, והמידע לא אמור להשתנות לאורך חיי המחלקה
  - ניתן לממש במו בל מחלקה רגילה או שניתן לעשות שימוש ב records

records

את ה final הזה נסביר בהמשך הקורס

```
public final class Rectangle {
        final float length;
        final float width;

        public Rectangle(float length, float width) {
                this.length = length;
                this.width = width;
        }

        public float length() { return length; }
        public float width() { return width; }
}
```

```
public record Rectangle(float length, float width) { }
```

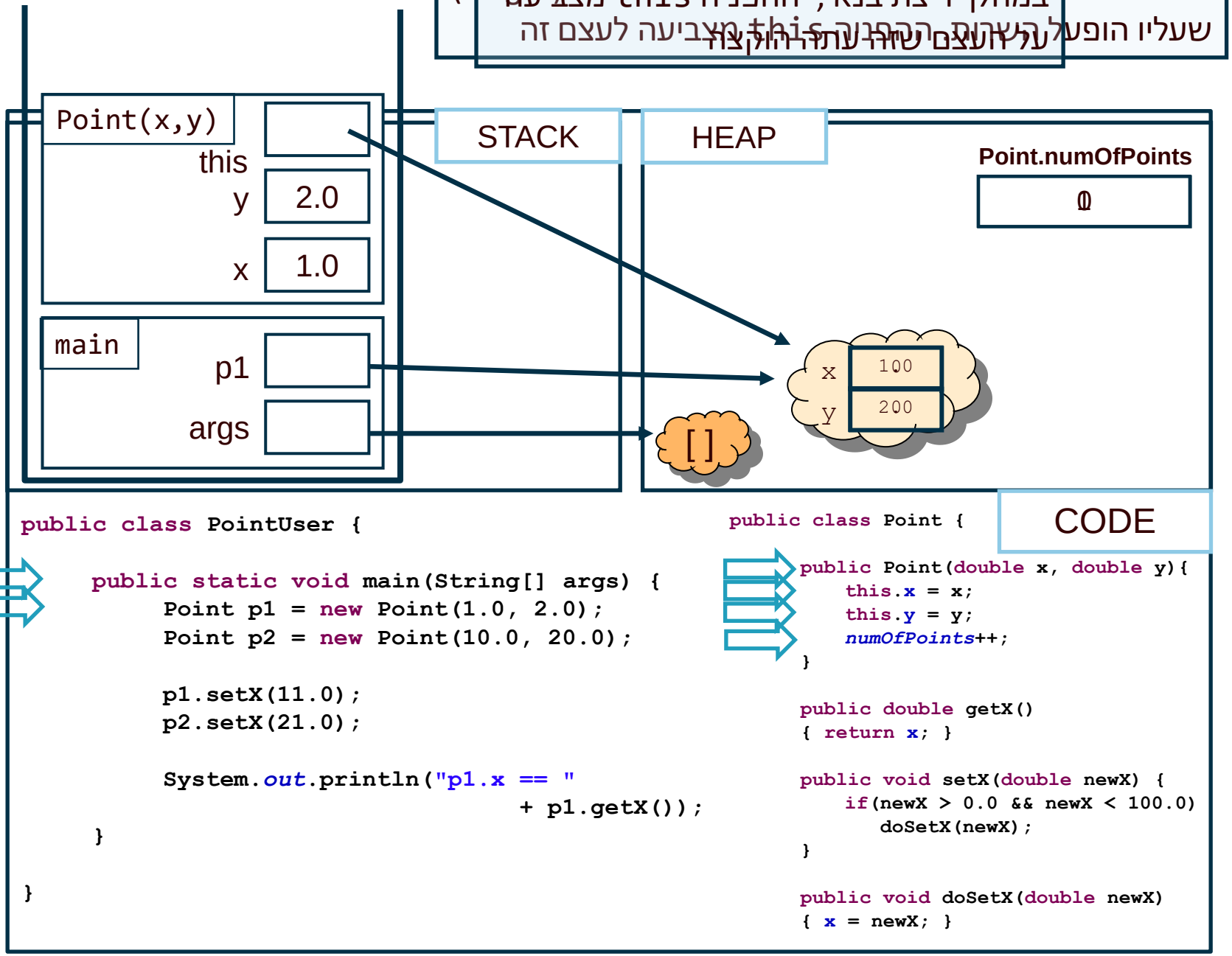
ניתן להוסיף מתודות ולהוסיף קוד לבנאי

```
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

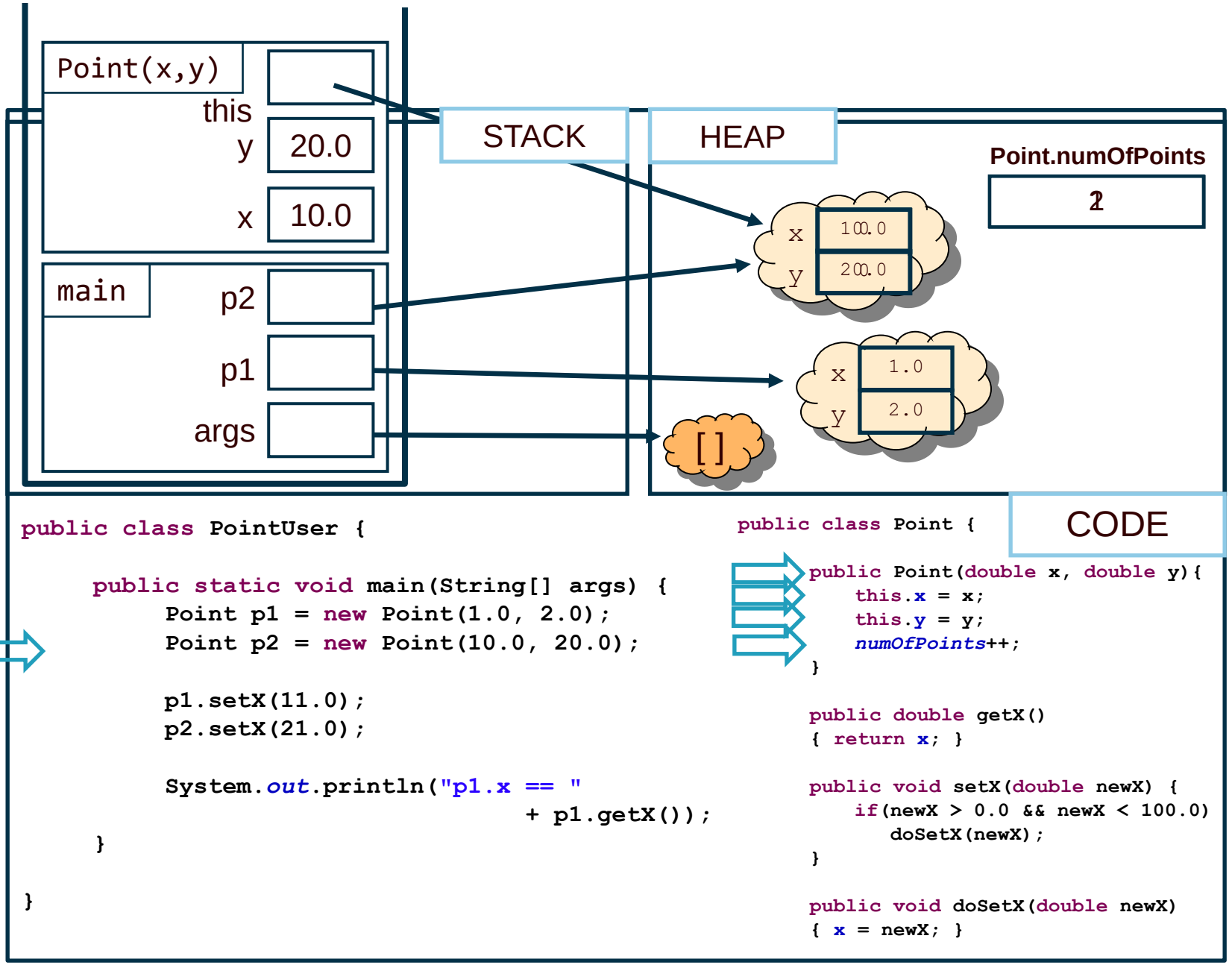
```
public class PointUser {

   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());
   }

}
```

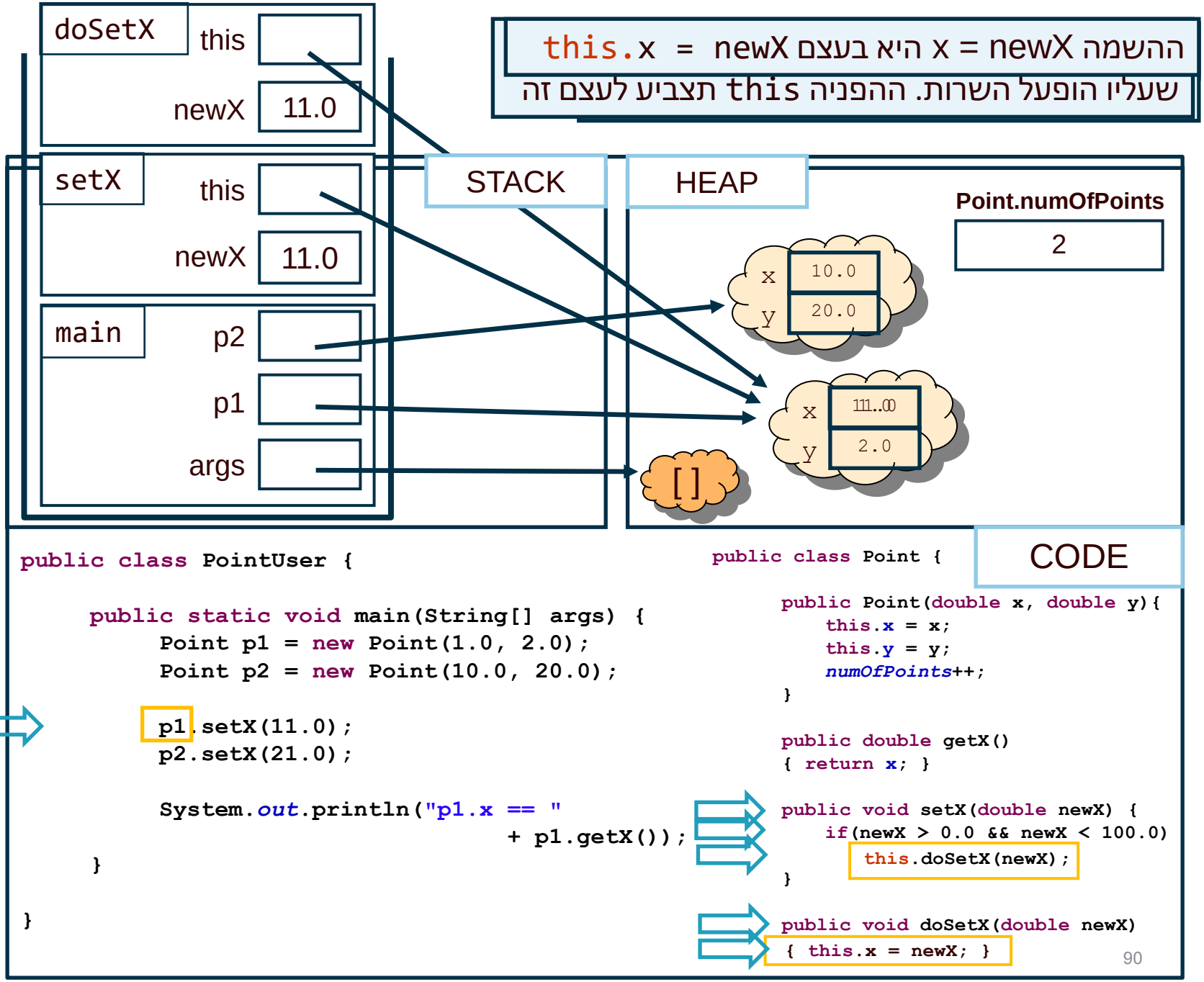
בכל הפעלה של שיטת מופע או בנאי יש עצם מטרה (target)
שעליו הופעל השרשור. ההפניה this מצביעה לעצם זה
צמחלף ויצרו בנאי, והפניית this מצב עם
על העצם שזה עתה הוקצה

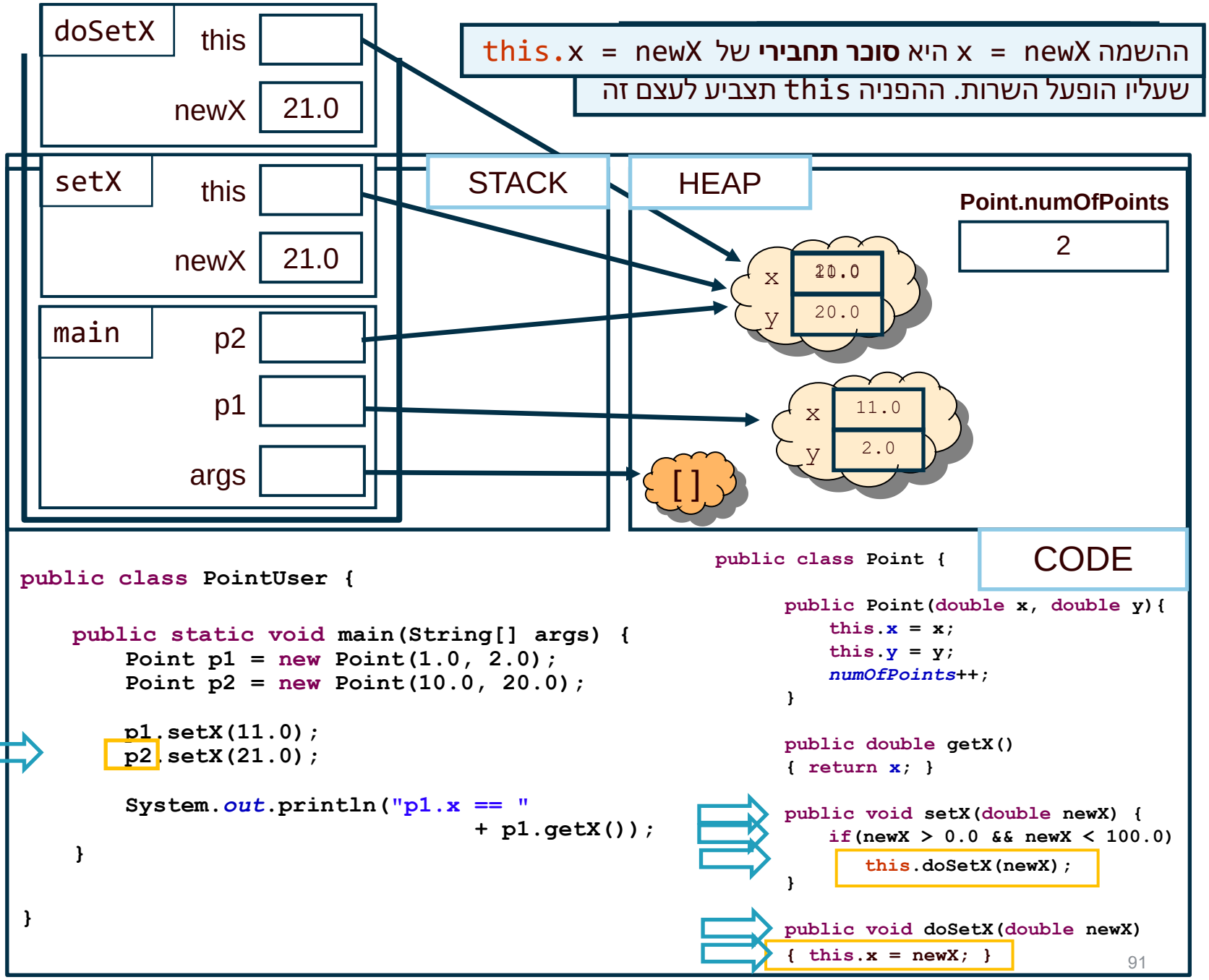



```
public class PointUser {

    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());
    }

}
```

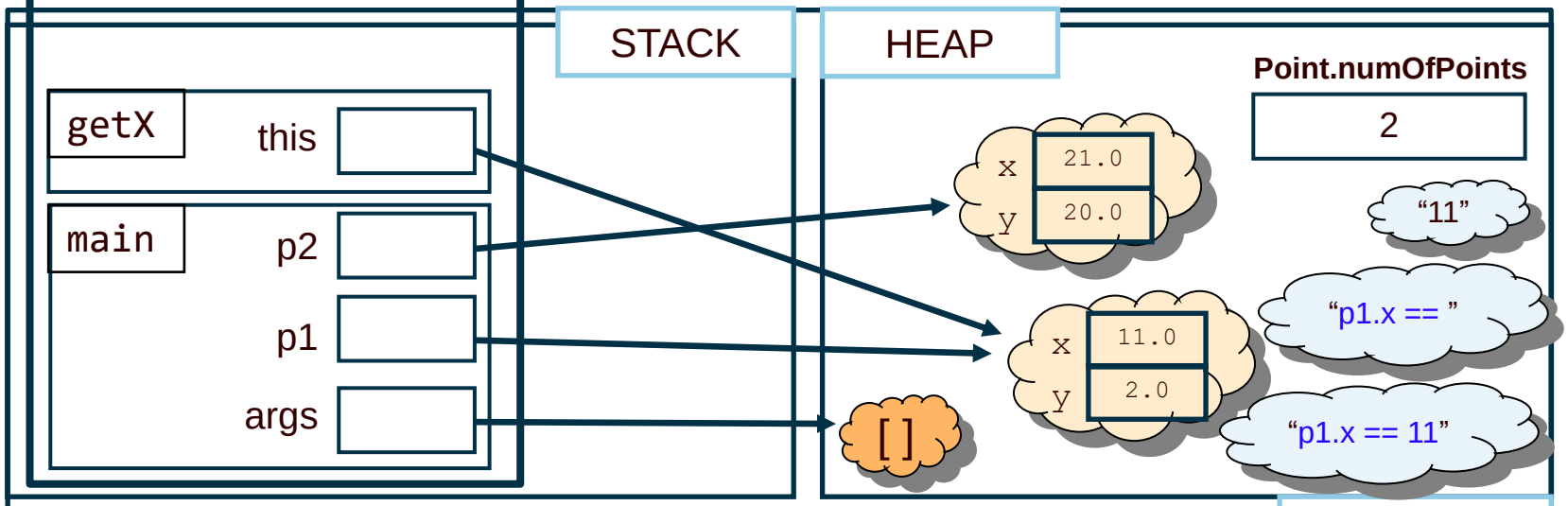
```
public class Point {

    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; }
```


Point(x,y)
this
y
20.0
x
10.0
main
p2
p1
args
STACK
HEAP
Point.numOfPoints
2
x
100.0
y
200.0
x
1.0
y
2.0
[ ]
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; }


doSetX
this
newX
11.0
setX
this
newX
11.0
main
p2
p1
args
STACK
HEAP
Point.numOfPoints
2
x
y
10.0
20.0
111..00
2.0
[]
ההשמה this.x = newX היא בעצם x = newX
שעליו הופעל השרות. ההפניה this תצביע לעצם זה
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; }


ההשמה newX = x היא סוכר תחבירי של this.x = newX
שעליו הופעל השרות. ההפניה this תצביע לעצם זה
doSetX
this
newX
21.0
setX
this
newX
21.0
main
p2
p1
args
STACK
HEAP
Point.numOfPoints
2
x
y
20.0
x
y
11.0
2.0
[]
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; }

"return this.x" המשפט return x יחזיר



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 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 גלובליים. קיים עותק אחד שלהם, שנוצר בעת טעינת קוד המחלקה לזכרון, ללא קשר ליצירת עצמים מאותה המחלקה

מה ראינו עד כה?

- תכנות ב Java - בסיס
- 8 הטיפוסים היסודיים ב Java, טיפוס המחרוזת וטיפוס המערך
- תחביר Java: פקודות תנאי, לולאות, אופרטורים ,switch
- שרותי מחלקה ומשתני מחלקה
- העמסה
- מודל הזיכרון של Java (המחסנית והערימה)
- מחלקות ושירותי מופע
- enum
- חוזים

טיפוסי מנייה enums

- טיפוסי מנייה הם טיפוסים להם קיים רק מספר קטן של ערכים אפשריים
- בלומר, כל מופעיהם קבועים וידועים מראש
- לדוגמה, נרצה להגדיר טיפוס שמייצג צבע אפשרי ברמזור: אדום, צהוב, וירוק, ואלו הערכים האפשריים היחידים לטיפוס זה

```
public enum TrafficLightColor {
   RED, YELLOW, GREEN;
}
```

- המופעים היחידים של טיפוס זה (בדוגמה זו יש 3) ניתנים לגישה דרך השדות הסטטיים:

```
TrafficLightColor.RED
TrafficLightColor.YELLOW
TrafficLightColor.GREEN
```

- **enum** התווסף לשפה בגירסה Java 5.0 (בשנת 2004)

דוגמה נוספת

- טיפוס שמייצג את ה"סדרה" של קלף

- ובך נוכל להגדיר טיפוס חדש של קלף

```
public enum Suit {
   SPADES,
   HEARTS,
   CLUBS,
   DIAMONDS;
}
```

```
public class PlayingCard {
  private Suit suit;
  private int rank;

  public PlayingCard(Suit suit, int rank) {
         this.suit = suit;
         this.rank = rank;
  }

  public Suit getSuit() {
         return suit;
  }
}
```

מה ראינו עד כה?

- תכנות ב Java - בסיס
- 8 הטיפוסים היסודיים ב Java, טיפוס המחרוזת וטיפוס המערך
- תחביר Java: פקודות תנאי, לולאות, אופרטורים ,switch
- שרותי מחלקה ומשתני מחלקה
- העמסה
- מודל הזיכרון של Java (המחסנית והערימה)
- מחלקות ושירותי מופע
- enum
- חוזים

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)

דוגמה:

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

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

- בדוגמה זו הפונקציה **main** היא **לקוחה** של הפונקציה **doSomething()**
- **doSomething** 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 doSomething(){
  // doing...
}
```

Supplier.java

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

Client.java

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

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

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...