תוכנה 1 בשפת Java

שיעור מספר 2: מערכים, העמסה

בית הספר למדעי המחשב

אוניברסיטת תל אביב

טיפוסים שאינם יסודיים ומערכים

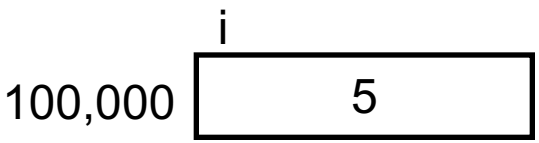
משתנים שאינם יסודיים (non primitive variables)

- פרט ל-8 הטיפוסים שסקרנו עד כה, כל שאר הטיפוסים ב Java אינם פרימיטיבים
- הספריה התקנית של Java מכילה יותר מ- 3000 טיפוסים (!) ואנו כמתכנתים נשתמש בהם ואף ניצור טיפוסים חדשים
- מערכים ומחרוזות אינם טיפוסים יסודיים, אולם מכיוון שאנו שנזדקק להם כבר בשיעורים הקרובים נדון בקצרה בטיפוסי הפנייה
- משתנה מטיפוס שאינו יסודי נקרא **הפנייה** (reference type)
  - לעיתים נשתמש בכינויים שקולים כגון: התייחסות, מצביע, מחוון, פוינטר
  - בשפות אחרות (למשל C++) יש הבדל בין המונחים השונים, אולם ב Java כולם מתייחסים למשתנה שאינו יסודי

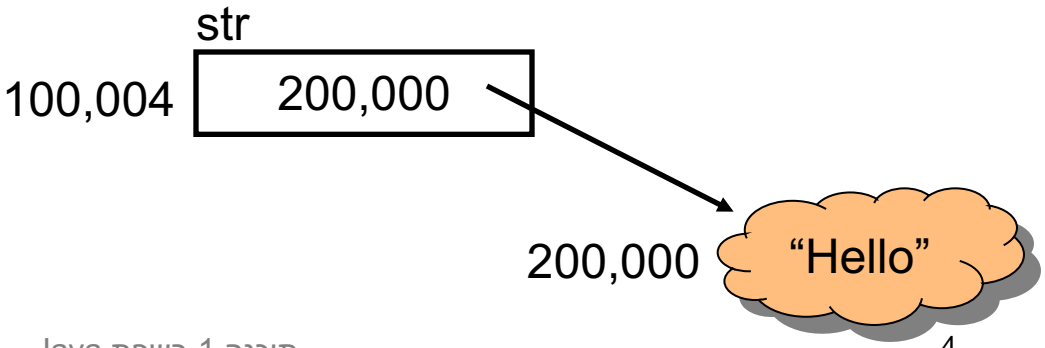
הפניות ומשתנים יסודיים

- ביצירת **משתנה מטיפוס יסודי** אנו יוצרים מקום בזיכרון בגודל ידוע שיכול להכיל ערך מטיפוס מסוים
- ביצירת **משתנה הפנייה** אנו יוצרים מקום בזכרון, שיכול להכיל כתובת של מקום אחר בזכרון 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)
- 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

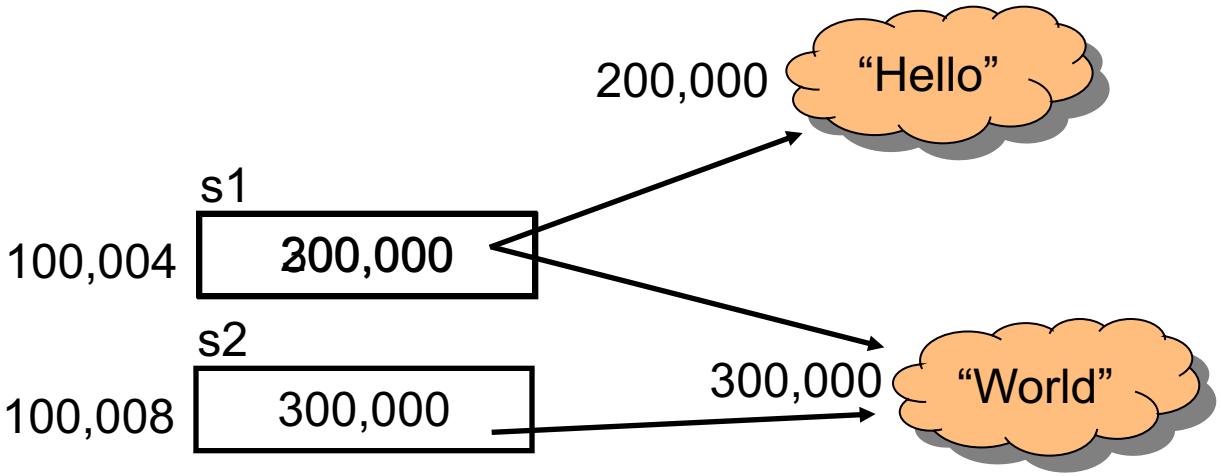
פעולות על הפניות

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

⇨ String s1 = “Hello”;

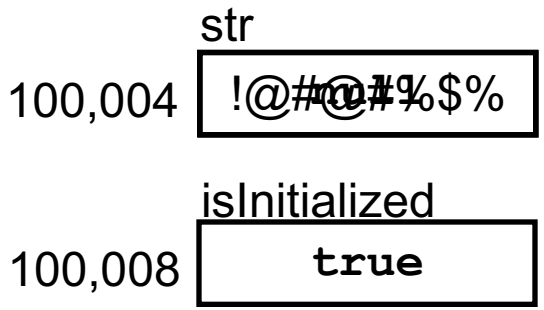
⇨ String s2 = “World”;

⇨ s1= s2;



null ערך

- 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;

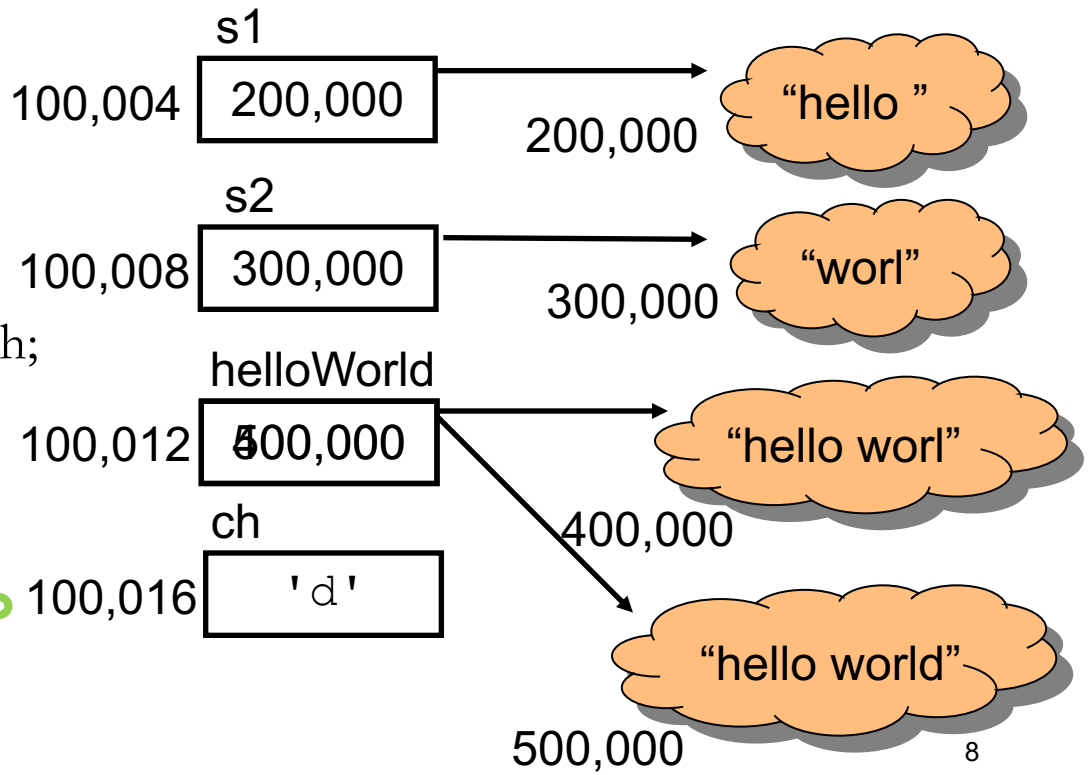
⇨ str = null;

⇨ boolean isInitialized = (str == null);

שרשור מחרוזות

- כאשר 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;
```



מה **באמת** כתוב בתא ch?

פניה לעצם המוצבע

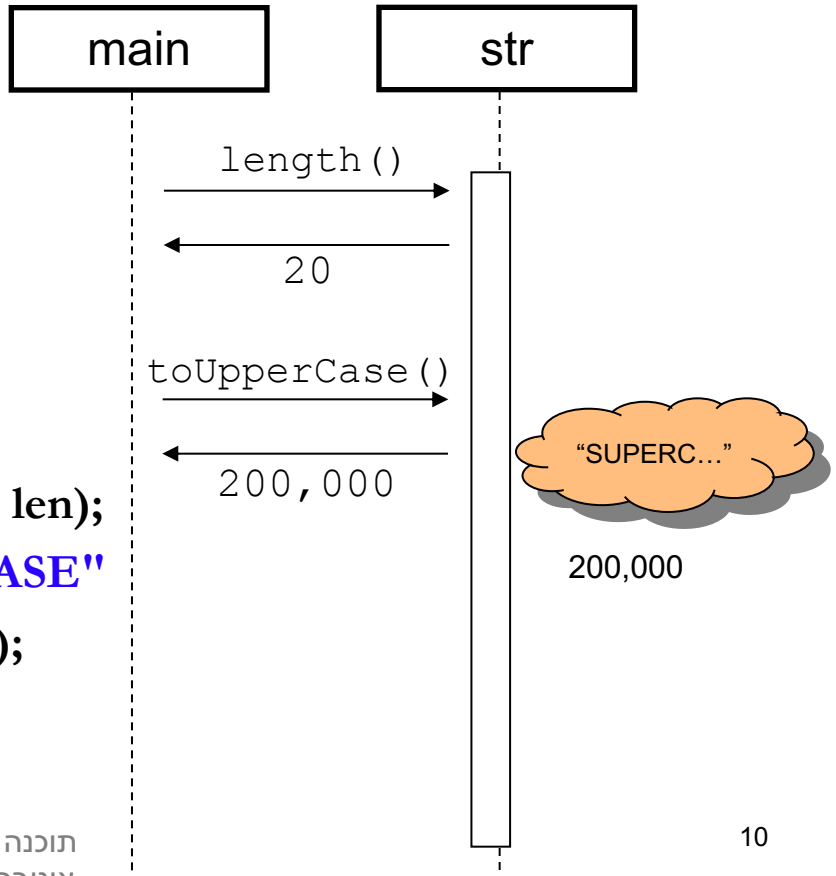
- עד עכשיו כל הפעולות שבצענו היו על ההפנייה. איך ניגשים לעצם המוצבע?
- אופרטור '.' (הנקודה) מאפשר גישה לעצם המוצבע
- מה עושים עם זה?
  - אפשר לבקש **בקשות**
  - אפשר לשאול **שאלות** (ולקבל תשובות)
  - לעיתים רחוקות אפשר לגשת **למאפיינים פנימיים** ישירות
- הבקשות השאלות והמאפיינים הפנימיים משתנים מעצם לעצם לפי טיפוסו (אם כי יש מספר קטן של בקשות שאפשר לבקש מכל עצם ב Java)

דוגמא

- בדוגמא הבאה נשאל עצם מחרוזת לאורכו, ואח"כ נבקש ממנו לייצר גירסת Uppercase של עצמו. לסיום נדפיס את התוצאות:

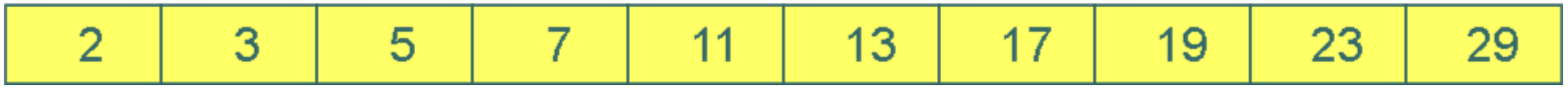
```
public class StringExample {

  public static void main(String[] args) {
    String str = "SupercaliFrajalistic";
    int len = str.length();
    String upper = str.toUpperCase();
    System.out.println("String length is " + len);
    System.out.println("String in UPPERCASE"
                       + "is '" + upper + "'");
  }
}
```



מערכים

- מייצג סדרת משתנים מאותו הטיפוס (בין אם פרימיטיבי או הפניה).
- לדוגמא: מערך של איברים מטיפוס int:

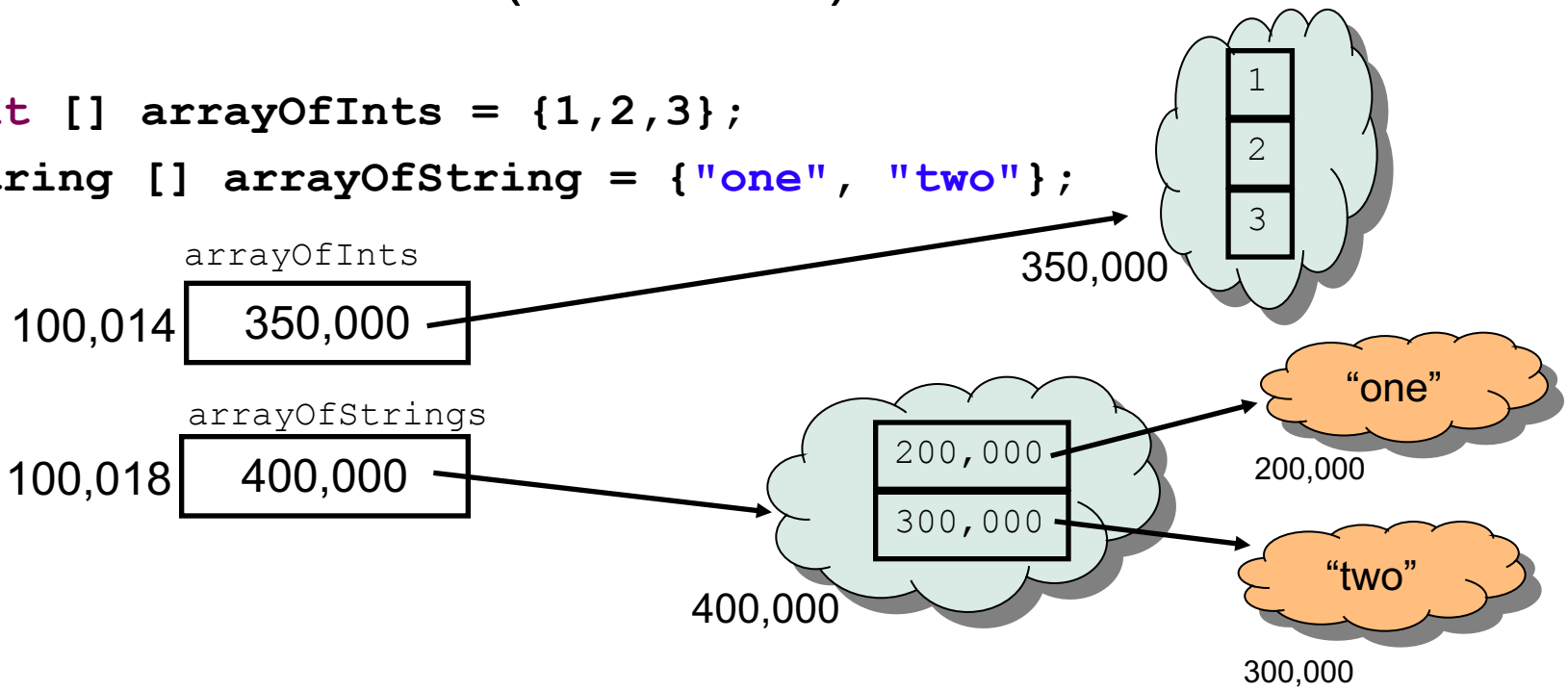


- תאים במערך יושבים בדרך כלל ברצף בזיכרון (Java רצה על מכונה וירטואלית!) כך שגישה סידרתית אליהם עשויה להיות יעילה
- מערך אינו זהה לטיפוס list שאתם מכירים ב Python!

מערכים

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

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



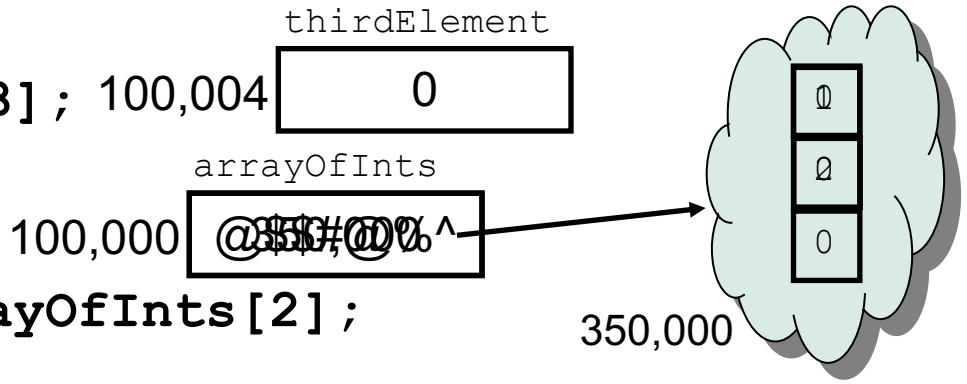
מערכים

- נשים לב להבדל בין מערכים 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];
```



- אברי מערך שהוקצה ע"י `new` מאותחלים אוטומטית לפי טיפוסם:
  - טיפוס הפנייה מאותחל ל- `null`
  - הטיפוסים הפרימיטיביים השלמים מאותחלים ל-0
  - הטיפוסים הפרימיטיביים הממשיים מאותחלים ל- 0.0
  - הטיפוסים הפרימיטיבי `boolean` מאותחלים ל- `false`
  - הטיפוסים הפרימיטיבי `char` מאותחל לתו שערך ה `Unicode` שלו הוא 0

ניתן לשאול מערך לאורכו

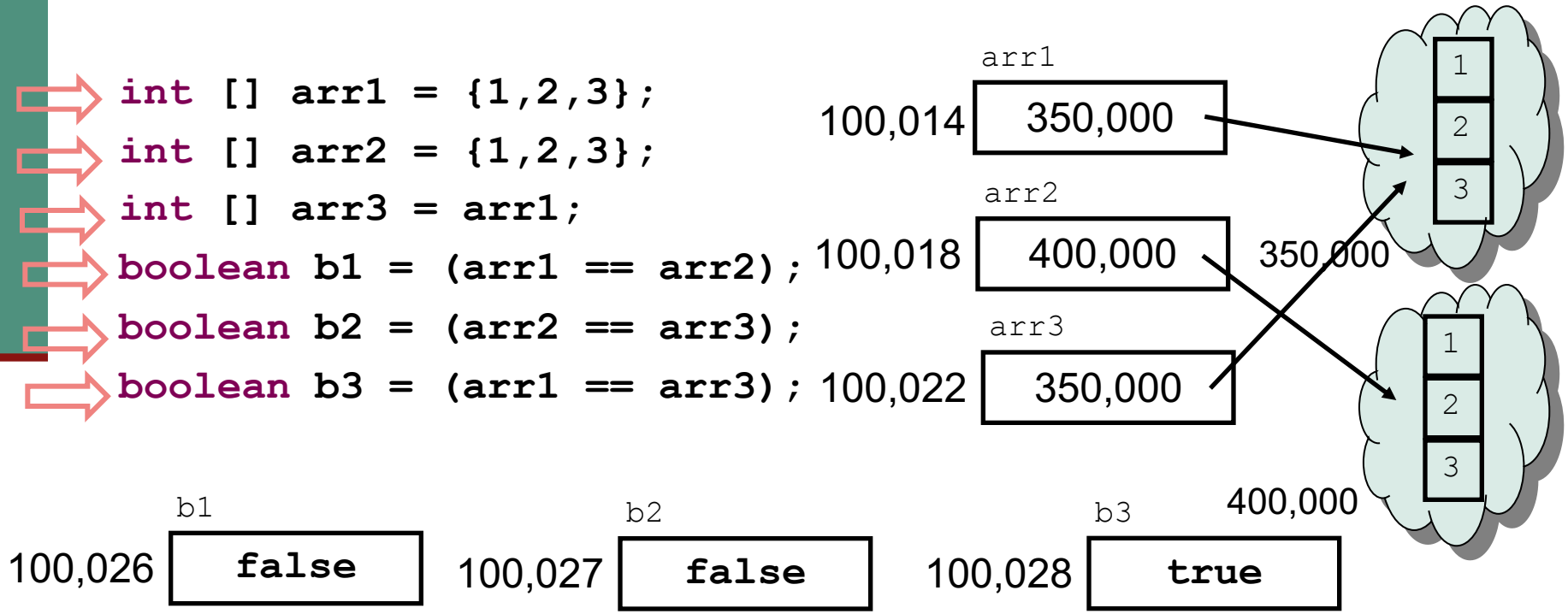
- אורכו של מערך, הוא מאפיין פנימי אשר ניתן לגשת אליו ישירות בעזרת אופרטור הנקודה

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

הפניות ואופרטור ההשוואה (==)

- אופרטור ההשוואה (==) כאשר הוא מופעל על משתני הפניה, משווה את ההפניות (הכתובות המופיעות בהן) ולא את העצמים המוצבעים:

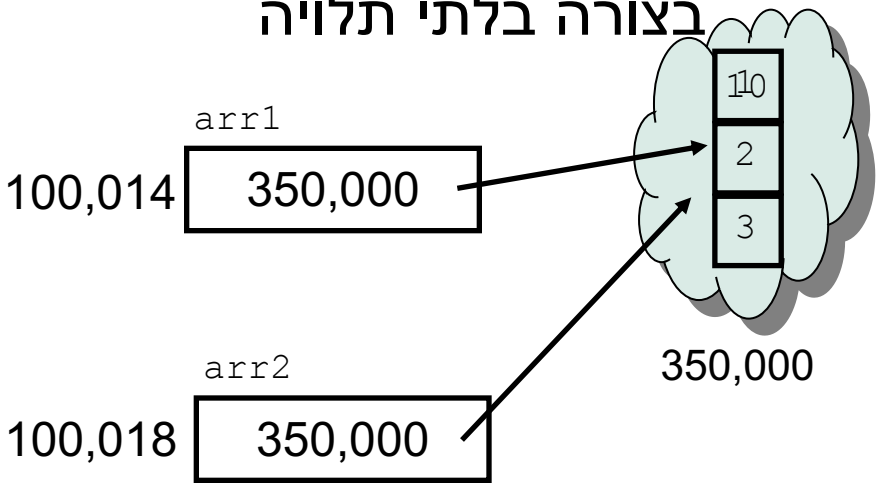
```
int [] arr1 = {1,2,3};
int [] arr2 = {1,2,3};
int [] arr3 = arr1;
boolean b1 = (arr1 == arr2);
boolean b2 = (arr2 == arr3);
boolean b3 = (arr1 == arr3);
```



שיתוף (sharing, aliasing)

- אם שתי הפניות מצביעות לאותו עצם, העצם הוא משותף לשתיהן. אין עותק נפרד לכל הפנייה
- כל אחת מההפניות יכולה לשנות את העצם המשותף המוצבע בצורה בלתי תלויה

```
int [] arr1 = {1,2,3};
int [] arr2 = arr1;
arr2[0] = 10;
System.out.println(arr1[0]);
// ? מה יודפס
```



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 (+)

שירותי מחלקה והעמסה

שרותי מחלקה (static methods)

- בדומה לשיגרה (פרוצדורה, פונקציה) בשפות תכנות אחרות, *שרות מחלקה* הוא סדרת משפטים שניתן להפעילה ממקום אחר בקוד של ג'אווה ע"י קריאה לשרות, והעברת אפס או יותר ארגומנטים

- שירותים כאלה מוכרזים 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**

החזרת ערך משרות ומשפט return

- משפט **return**:

```
return <optional-expression>;
```

- ביצוע משפט **return** מחשב את הביטוי (אם הופיע), מסיים את השרות המתבצע כרגע וחוזר לנקודת הקריאה
- אם המשפט כולל ביטוי ערך מוחזר, ערכו הוא הערך שהקריאה לשרות תחזיר לקורא
- טיפוס הביטוי צריך להיות תואם לטיפוס הערך המוחזר של השרות
- אם טיפוס הערך המוחזר מהשרות הוא **void** , משפט ה- **return** לא יכלול ביטוי, או שלא יופיע משפט **return** כלל, והשרות יסתיים כאשר הביצוע יגיע לסופו

דוגמא לשרות המחזיר ערך

```
public static int arraySum(int [] arr) {
   int sum = 0;
   for (int i : arr) {
      sum += i;
   }
   return sum;
}
```

- אם שרות מחזיר ערך, כל המסלולים האפשריים של אותו שרות (flows) צריכים להחזיר ערך
- איך נתקן את השרות הבא:

```
public static int returnZero() {
   int one = 1;
   int two = 2;

   if (two > one)
      return 0;
}
```

גוף השרות

- גוף השרות מכיל הצהרות על משתנים מקומיים (variable declaration) ופסוקים ברי ביצוע (כולל return)
- **משתנים מקומיים** נקראים גם משתנים זמניים, משתני מחסנית או משתנים אוטומטיים
- הצהרות יכולות להכיל פסוק איתחול בר ביצוע (ולא רק אתחול ע"י ליטרלים)

```
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

יש צורך באתחול מפורש

- קטע הקוד הבא, לדוגמא, אינו עובר קומפילציה:

```
int i;
int one = 1;

if (one == 1)
   i = 0;

System.out.println("i=" + i);
```

הקומפיילר צועק ש- `i` עלול שלא להיות מאותחל לפני השימוש בו

קריאה לשרות (method call)

- קריאה לשרות (לפעמים מכונה – "זימון מתודה") שאינו מחזיר ערך (טיפוס הערך המוחזר הוא void) תופיע בתור משפט (פקודה), ע"י ציון שמו וסוגריים עם או בלי ארגומנטים

- קריאה לשרות שמחזיר ערך תופיע בדרך כלל כביטוי (למשל בצד ימין של השמה, כחלק מביטוי גדול יותר, או כארגומנט המועבר בקריאה אחרת לשרות)

- קריאה לשרות שמחזיר ערך יכולה להופיע בתור משפט, אבל יש בזה טעם רק אם לשרות תוצאי לוואי, כי הערך המוחזר הולך לאיבוד

- גם אם השרות אינו מקבל ארגומנטים, יש חובה לציין את הסוגריים אחרי שם השרות

```
public class MethodCallExamples {

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

    public static int arraySum(int[] arr) {
        int sum = 0;
        for (int i : arr)
            sum += i;
        return sum;
    }

    public static void main(String[] args) {

        printMultipleTimes("Hello", 5);

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

הגדרת שרות void (פרוצדורה)

הגדרת שרות עם ערך מוחזר (פונקציה)

קריאה לשרות

קריאה לשרות

```
public class MethodCallExamples {

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

    public static int arraySum(int[] arr) {
        int sum = 0;
        for (int i : arr)
            sum += i;
        return sum;
    }

    public static void main(String[] args) {

        printMultipleTimes("Hello", 5);

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

    }
}
```

ניתן לוותר על משתנה העזר sumOfPrimes

```
public class MethodCallExamples {

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

    public static int arraySum(int[] arr) {
        int sum = 0;
        for (int i : arr)
            sum += i;
        return sum;
    }

    public static void main(String[] args) {

        printMultipleTimes("Hello", 5);

        int[] primes = { 2, 3, 5, 7, 11 };
        arraySum(primes);
    }
}
```

אין חובה לקלוט את הערך
המוחזר משרות,
אולם אז הוא הולך לאיבוד

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

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**
- כבר השתמשנו בפונקציה מועמסת – `println` עבדה גם כשהעברנו לה משתנה פרימיטיבי וגם כשהעברנו לה מחרוזת
- נציג שלוש סיבות לשימוש בתכונת ההעמסה
  - נוחות
  - ערכי ברירת מחדל לארגומנטים
  - תאימות אחורה

העמסת פונקציות (שיקולי נוחות)

- נממש את max המקבלת שני ארגומנטים ומחזירה את הגדול מביניהם
- ללא שימוש בתכונת ההעמסה (בשפת C למשל) היה צורך להמציא שם נפרד עבור כל אחת מהמתודות:

```
public class MyUtils {
  public static double max_double(double x, double y){
     ...
  }

  public static long max_long(long x, long y){
     ...
  }
}
```

- השמות מלאכותיים ולא נוחים

העמסת פונקציות (פחות נוחות)

- בעזרת מנגנון ההעמסה ניתן להגדיר:
  - `public static double max(double x, double y)`
  - `public static long max(long x, long y)`
- בחלק מהמקרים, אנו משלמים על הנוחות הזו באי בהירות
- למשל, איזו מהפונקציות תופעל במקרים הבאים:

```
max(1L , 1L);      // max(long,long)
max(1.0 , 1.0);    // max(double,double)
max(1L , 1.0);     // max(double,double)
max(1 , 1);        // max(long,long)
```

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.
- ר' פרק 5, **Conversions and Promotions** של ה – JLS:
  http://docs.oracle.com/javase/specs/jls/se8/html/index.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 וערכי ברירת מחדל לארגומנטים

- ב Python ניתן להגדיר פונקציות עם פרמטרים עם ערכי ברירת מחדל. למשל:

```
def func(x, y=1):
        return x+y
```

לפונקציה func ניתן לקרוא בשתי צורות: עם פרמטר יחיד ועם שני פרמטרים.

- ב Java לא ניתן להגדיר ערכי ברירת מחדל. במקום זאת, נשתמש בהעמסה:

```
public static int func(int x){
        return func(x, 1);
}

public static int func(int x, int y){
        return x + y;
}
```

העמסה ותאימות לאחור

- נניח כי במערכת כלשהי כבר קיימת הפונקציה השימושית compute המבצעת חישוב כלשהו על הארגומנט x.

```
public static int compute(int x)
```

- לאחר זמן מה, כאשר הקוד כבר בשימוש במערכת (גם מתוך מחלקות אחרות שלא אתה כתבת), עלה הצורך לבצע חישוב זה גם בבסיסי ספירה אחרים (החישוב המקורי בוצע בבסיס עשרוני)

- בשלב זה לא ניתן **להחליף** את חתימת הפונקציה להיות:

```
public static int compute(int x, int base)
```

מכיוון שקטעי הקוד המשתמשים בפונקציה יפסיקו להתקמפל

העמסה, תאימות לאחור ושכפול קוד

- על כן במקום **להחליף** את חתימת השרות **נוסיף** פונקציה חדשה כגירסה מועמסת
  - משתמשי הפונקציה המקורית לא נפגעים
  - משתמשים חדשים יכולים לבחור האם לקרוא לפונקציה המקורית או לגרסה החדשה ע"י העברת מספר ארגומנטים מתאים

- בעיה – קיים דמיון רב בין המימושים של הגרסאות המועמסות השונות (גוף המתודות compute)
- דמיון רב מדי - שכפול קוד זה הינו בעייתי מאוד

שכפול קוד הוא הדבר הנורא ביותר בעולם (התוכנה)!

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
- כדי 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);
}
```

העמסת מספר כלשהו של ארגומנטים

- נניח שברצוננו לכתוב פונקציה שמחזירה את ממוצע הארגומנטים שקיבלה:

```
public static double average(double x){
   return x;
}

public static double average(double x1, double x2){
   return (x1 + x2) / 2;
}

public static double average(double x1, double x2, double x3){
   return (x1 + x2 + x3) / 3;
}
```

- למימוש 2 חסרונות:
  - שכפול קוד
  - לא תומך בממוצע של 4 ארגומנטים

העמסת מספר כלשהו של ארגומנטים

- רעיון: הארגומנטים יועברו כמערך

```
public static double average(double [] args) {
   double sum = 0.0;
   for (double d : args) {
      sum += d;
   }
   return sum / args.length;
}
```

- **יתרון**: שכפול הקוד נפתר
- **חסרון**: הכבדה על הלקוח - כדי לקרוא לפונקציה יש ליצור מערך

```
public static void main(String[] args) {
   double [] arr = {1.0, 2.0, 3.0};
   System.out.println("Averge is:" + average(arr));
```

ג'אווה באה לעזרה

- ב Java קיים תחביר להגדרת שרות עם **מספר לא ידוע** של ארגומנטים (vararg)
- תחביר מיוחד של שלוש נקודות (...) יוצר את המערך מאחורי הקלעים:

```
public static double average(double ... args) {
   double sum = 0.0;
   for (double d : args) {
      sum += d;
   }
   return sum / args.length;
}
```

- ניתן כעת להעביר לשרות מערך או ארגומנטים בודדים:

```
double [] arr = {1.0, 2.0, 3.0};
System.out.println("Averge is:" + average(arr));
System.out.println("Averge is:" + average(1.0, 2.0, 3.0));
```

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), מוכרים 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)

נשתמש במשתנה גלובלי `counter` כדי 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 ע"י ציון המשתנה כ **`final`**
- למשתנה שהוא **`final`** ניתן לבצע השמה פעם אחת בדיוק.
  כל השמה נוספת לאותו משתנה תגרור שגיאת קומפילציה
- דוגמא:

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

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

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