תוכנה 1 בשפת Java

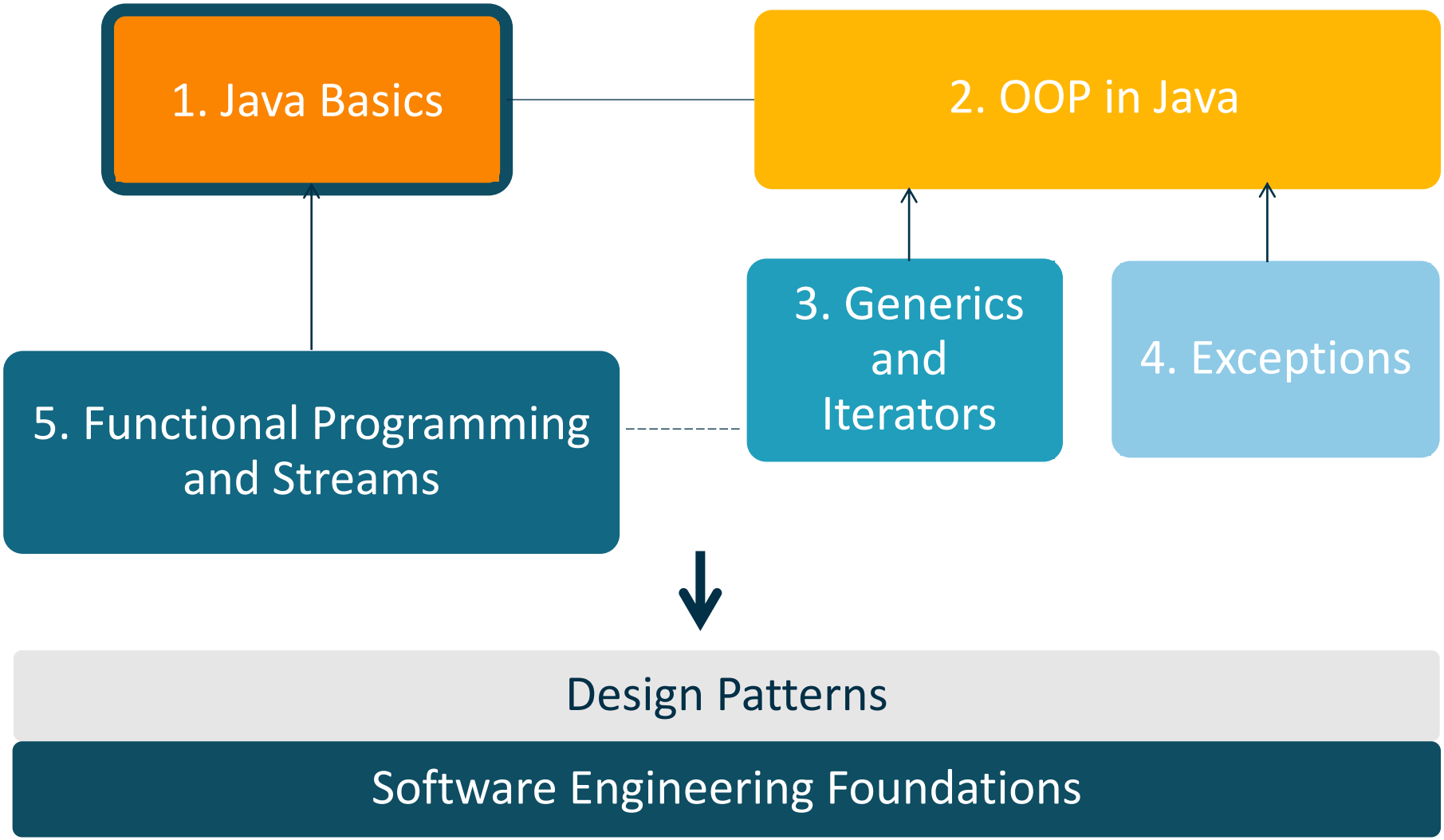
שיעור מספר 1: "שלום עולם"

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

התכנית להיום

טעימה משפת Java

פונקציית `main`

הכרות בסיסית עם git

תכנות בסיסי ב Java - בביתה ובעבודה עצמית 1:
8 הטיפוסים היסודיים, ביטויים ואופרטורים, טיפוסים שאינם יסודיים, טיפוס המחרוזת וטיפוס המערך

התכנית להיום

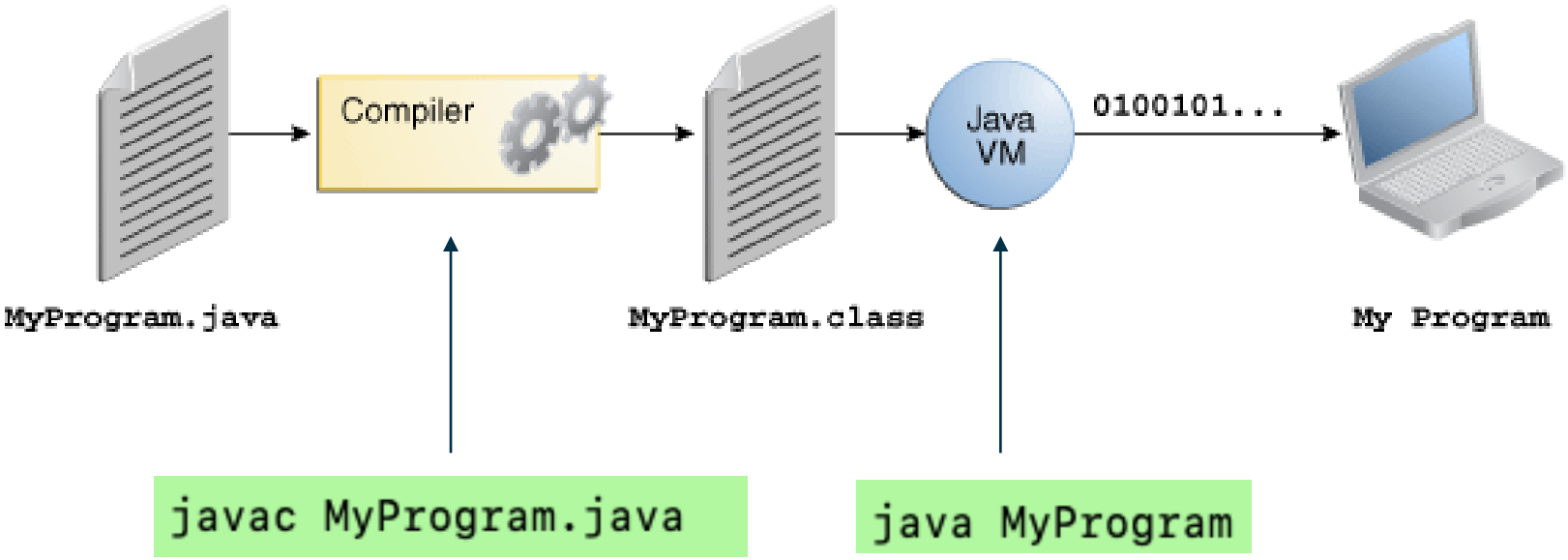
טעימה משפת Java

פונקציית `main`

הכרות בסיסית עם git

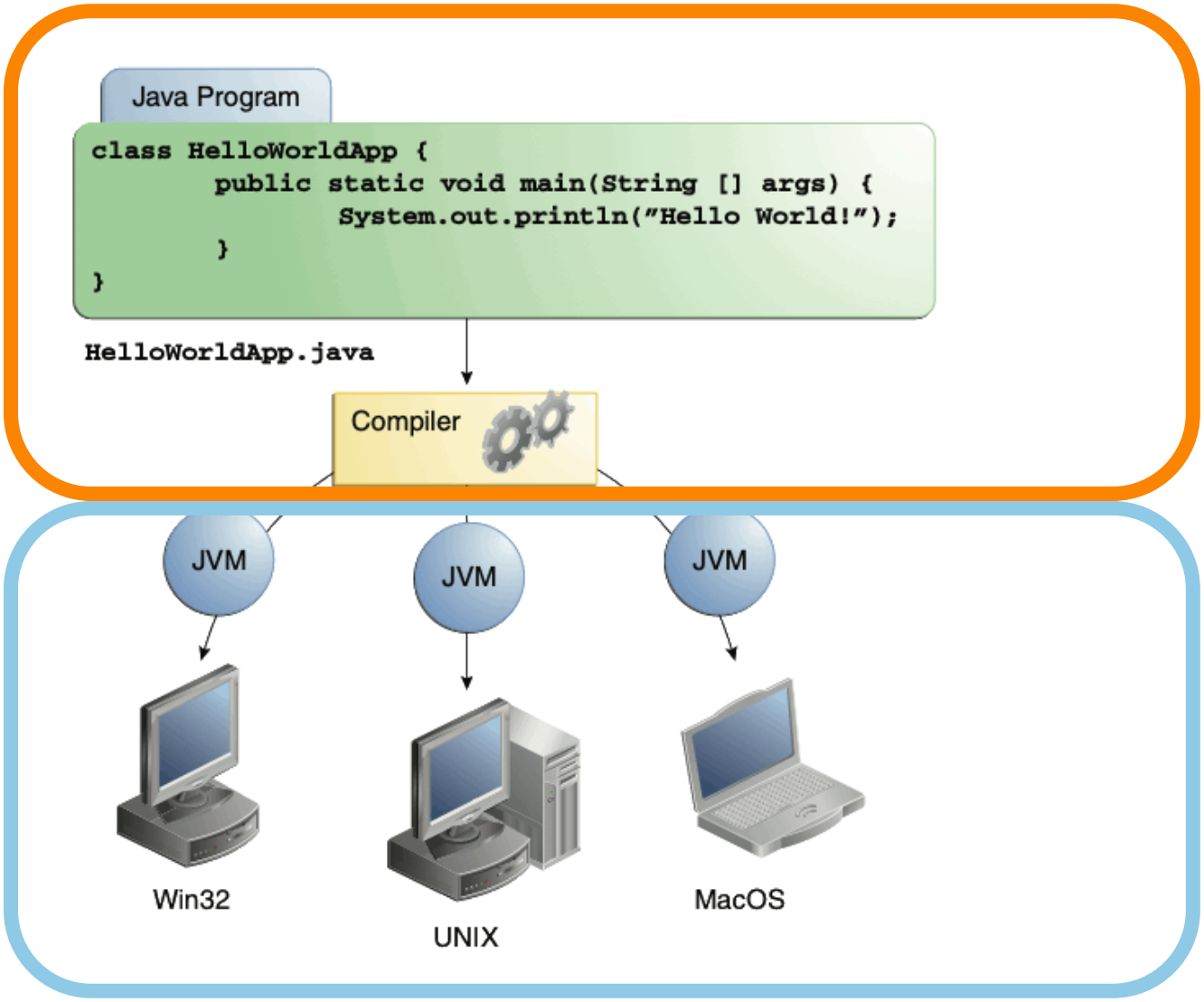
תכנות בסיסי ב Java - בביתה ובעבודה עצמית 1:
8 הטיפוסים היסודיים, ביטויים ואופרטורים, טיפוסים שאינם יסודיים, טיפוס המחרוזת וטיפוס המערך

שלום עולם



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

שלום עולם



שלב הפיתוח

שלב הריצה

המפרש (interpreter)

- את הקוד שנכתב בשפת Java מריץ *מפרש*
  - בדומה לשפת Python

- לריצה בעזרת מפרש יש כמה חסרונות:
  - מאט את מהירות הריצה
  - טעויות מתגלות רק בזמן הריצה

- לצורך כך הוסיפו ב Java שלב נוסף – *הידור* (compilation)

המהדר (compiler)

- מבצע עיבוד מקדים של קוד התוכנית (שכתובה בקובץ טקסט רגיל) ויוצר קובץ חדש בפורמט *נוח יותר*

- קובץ זה אינו קריא למתכנת אנושי (אף שניתן לפתוח אותו בעורך טקסט בגון Notepad), אולם המבנה שלו מותאם לקריאה ע"י המפרש של Java

- פורמט זה נקרא byte code והוא נשמר בקובץ עם סיומת .class

- בתהליך העיבוד ("קומפילציה") נבדק התחביר של הקוד – והשגיאות המתגלות מדווחות למתכנת

יבילות (portability)

- מדוע אנו מסתפקים בפורמט "נוח יותר"?
- מדוע אין המהדר יוצר קובץ בפורמט התואם *בדיוק* לחומרת המחשב, וכך היה נחסך בזמן ריצה גם שלב ה"הבנה" של הקוד?

הסיבה:

- איננו יודעים מראש על איזה מחשב בדיוק תרוץ תוכנית ה- Java שכתבנו
- תוכניות Java *חוצות סביבות* (cross platform)
  - סביבה = חומרה + מערכת הפעלה
  - תוכנית שנכתבה והודרה במחשב מסוים, תוכל לרוץ בכל מחשב אשר מותקן בו מפרש ל- Java

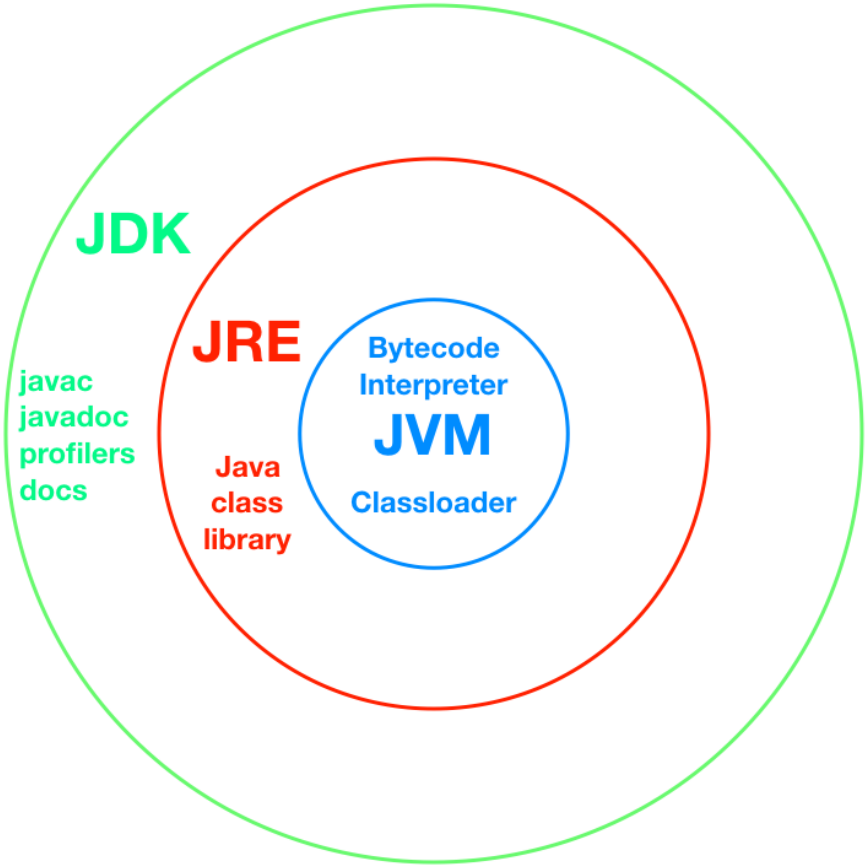
המכונה המדומה (**J**ava **V**irtual **M**achine- **JVM**)

- 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

איך הכל מתחבר?



https://purelyfunctional.tv/guide/clojure-jvm/#abbreviations

הקלות

- החל מגירסה Java10 ניתן להריץ תוכנית אשר בנויה מקובץ Java יחיד ללא צורך בשלב הקומפילציה.
  - ההרצה נעשית ע"י הפקודה java על קובץ המקור (source file) של התוכנית.
  - כל בדיקות הקומפילציה מבוצעות בעת הרצת הפקודה java אך לא נוצר קובץ class.
  - אם קיימת שגיאת קומפילציה בקוד, התוכנית לא תרוץ ונקבל פלט המתאר את השגיאה.
  - למה זה טוב?
    - המוטיבציה, לפחות על פי התיעוד הרשמי – להקל על מתכנתים אשר רק מתחילים להתמצא בשפה
    - למתקדמים – ניתן להריץ תוכניות Java הבנויות מקובץ יחיד בפורמט של סקריפטים (דוגמא בשקף הבא)

(להעשרה בלבד)

בקובץ hello:

```
#!/usr/bin/java --source 17

public class HelloWord {
        public static void main(String[] args) {
                System.out.println("Hello World");
        }
}
```

לא קוד ב java

קוד ב java

הרצה:

```
$ ./hello
Hello World
```

התכנית להיום

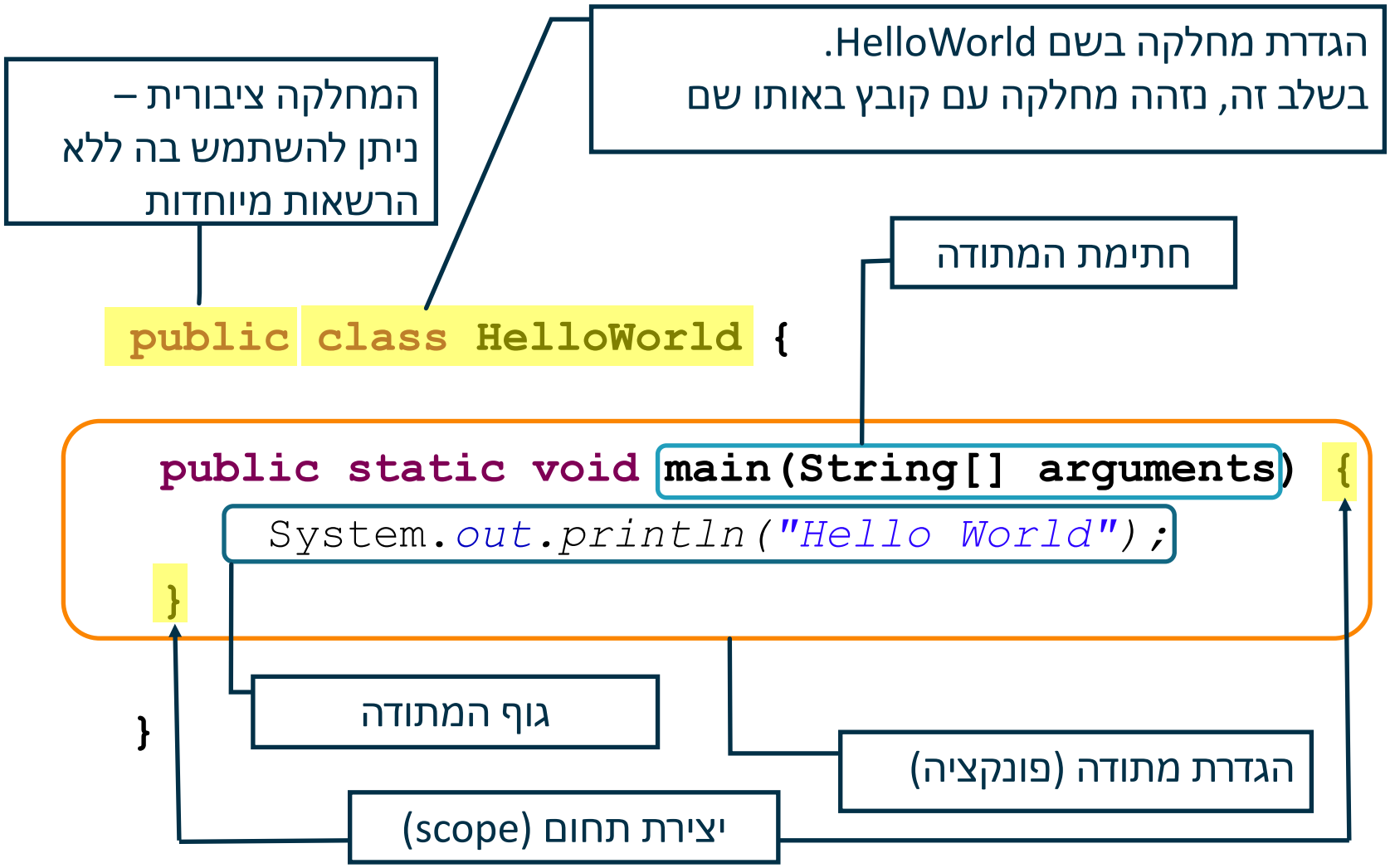
טעימה משפת Java

פונקציית main

הכרות בסיסית עם git

תכנות בסיסי ב Java - בביתה ובעבודה עצמית 1:
8 הטיפוסים היסודיים, ביטויים ואופרטורים, טיפוסים שאינם יסודיים, טיפוס המחרוזת וטיפוס המערך

שלום עולם


הגדרת מחלקה בשם HelloWorld.
בשלב זה, נזהה מחלקה עם קובץ באותו שם
המחלקה ציבורית –
ניתן להשתמש בה ללא
הרשאות מיוחדות
חתימת המתודה
public class HelloWorld {
public static void main(String[] arguments) {
System.out.println("Hello World");
}
}
גוף המתודה
הגדרת מתודה (פונקציה)
יצירת תחום (scope)

המתודה main

```
public static void main(String[] arguments) {
    System.out.println("Hello World");
}
```

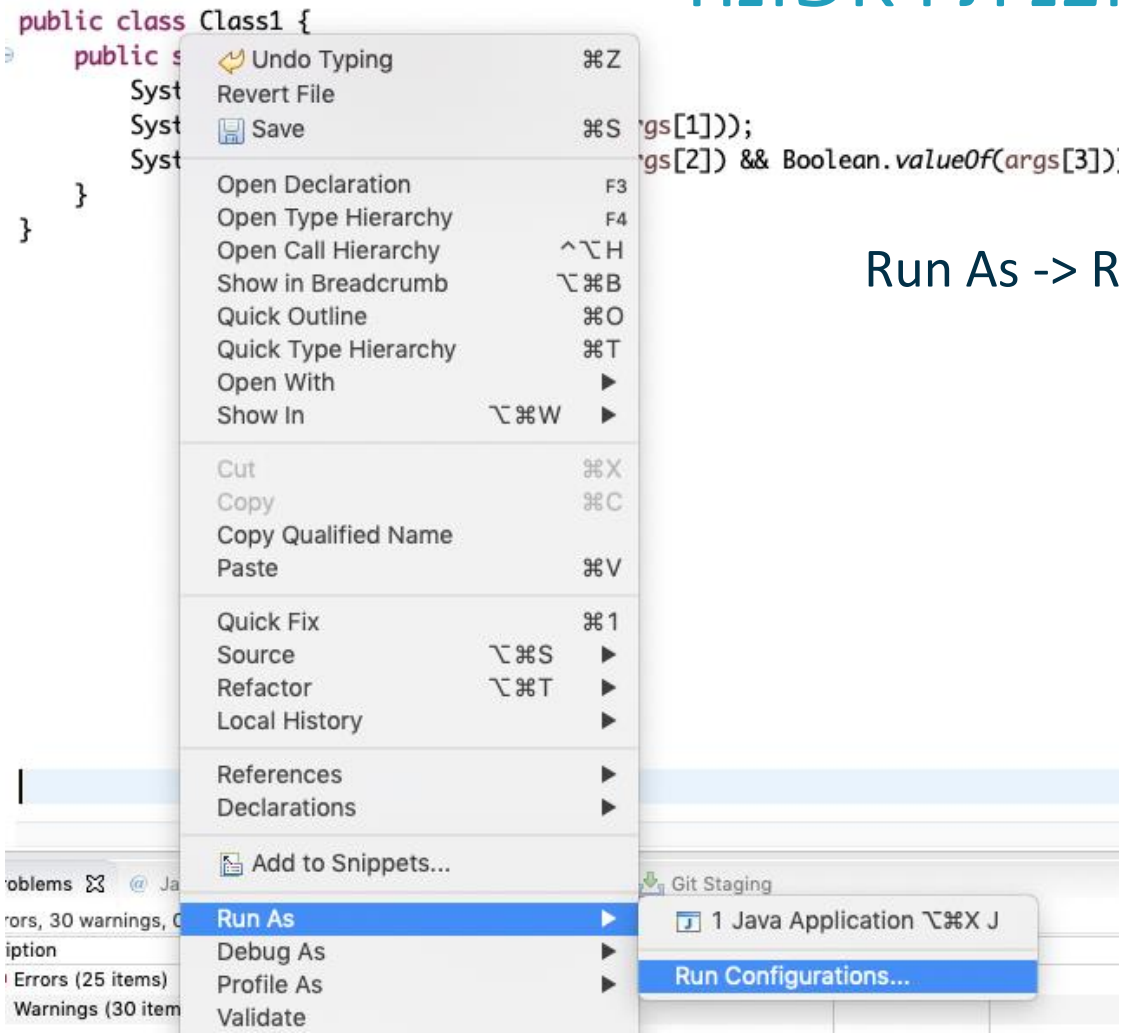
- כאשר אנו מריצים מחלקה ה JVM מחפש מתודה המוגדרת באופן הבא:
  - `main` – שם המתודה
  - `public` - המתודה ציבורית – ניתן להשתמש בה ללא הרשאות מיוחדות
  - `static` – מתודה של המחלקה (יוסבר בהמשך)
  - `void` – טיפוס הערך המוחזר. למתודה זו אין ערך מוחזר (ריק = void)

תוכנית ראשונה

```
public class Class1 {
   public static void main(String[] args){
      System.out.println(args[0]);
      System.out.println(1 + Integer.parseInt(args[1]));
      System.out.println(Boolean.parseBoolean(args[2]) &&
                                          Boolean.parseBoolean(args[3]));
   }
}
```

- התוכנית מבצעת 3 הדפסות. של מה?
- פעולות שאנחנו מבצעים על הארגומנטים של התוכנית
- מאיפה מגיעים הארגומנטים?

תוכנית ראשונה



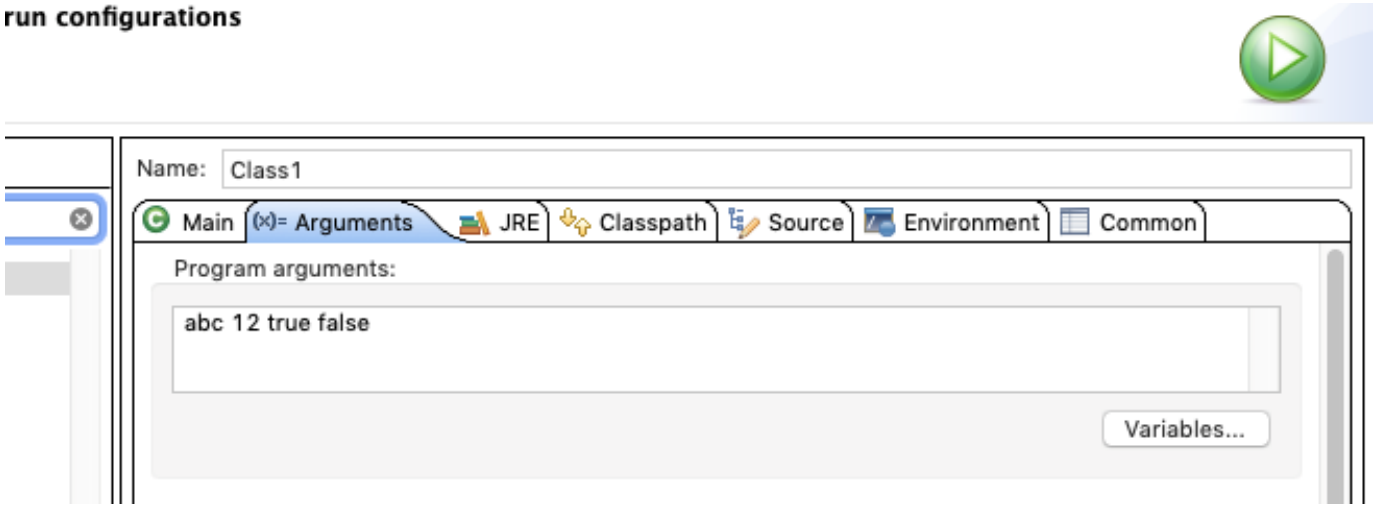
- ב eclipse:

1. Run As -> Run Configuration

תוכנית ראשונה

- ב eclipse:

1. Run As -> Run Configuration
2. Arguments



תוכנית ראשונה

- ב eclipse:

1. Run As -> Run Configuration
2. Arguments

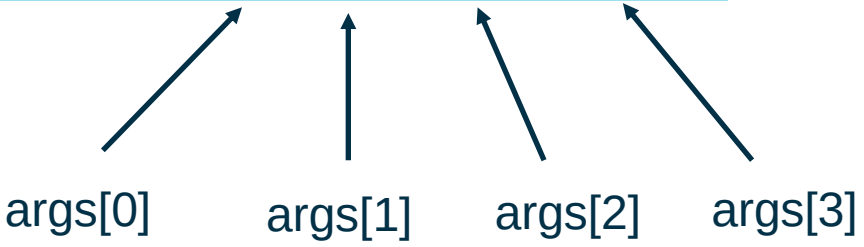
- בשורת הפקודה:

```
java Class1 abc 12 true false
```

תוכנית ראשונה

```
public class Class1 {
   public static void main(String[] args){
      System.out.println(args[0]);
      System.out.println(1 + Integer.parseInt(args[1]));
      System.out.println(Boolean.parseBoolean(args[2]) &&
                                          Boolean.parseBoolean(args[3]));
   }
}
```

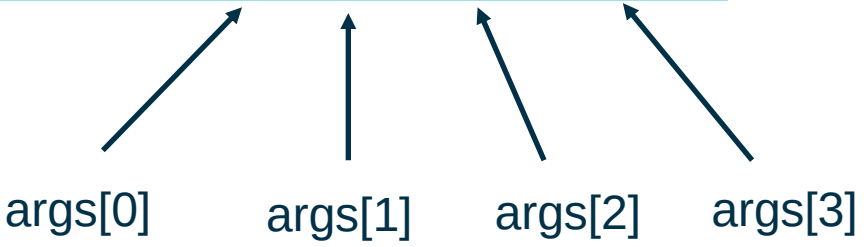
```
java Class1 abc 12 true false
```



תוכנית ראשונה

```
public class Class1 {
   public static void main(String[] args){
     System.out.println(args[0]);
      System.out.println(1 + Integer.parseInt(args[1]));
      System.out.println(Boolean.parseBoolean(args[2]) &&
                                          Boolean.parseBoolean(args[3]));
   }
}
```

```
java Class1 abc 12 true false
```

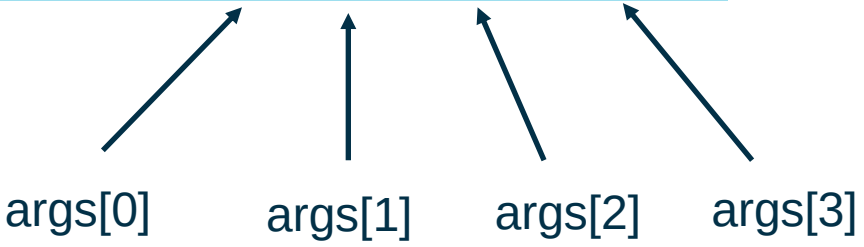


הדפסה של args[0] ללא שום פעולות נוספות

תוכנית ראשונה

```
public class Class1 {
   public static void main(String[] args){
      System.out.println(args[0]);
      System.out.println(1 + Integer.parseInt(args[1]));
      System.out.println(Boolean.parseBoolean(args[2]) &&
                                          Boolean.parseBoolean(args[3]));
   }
}
```

```
java Class1 abc 12 true false
```

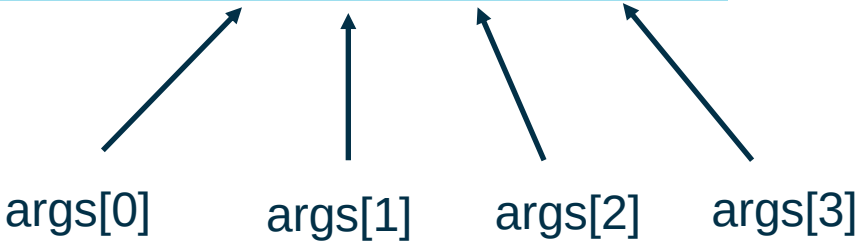


ממירים את args[1] מ string ל int, ומוסיפים לו 1.

תוכנית ראשונה

```
public class Class1 {
   public static void main(String[] args){
      System.out.println(args[0]);
      System.out.println(1 + Integer.parseInt(args[1]));
      System.out.println(Boolean.parseBoolean(args[2]) &&
                                    Boolean.parseBoolean(args[3]));
   }
}
```

```
java Class1 abc 12 true false
```



ממירים את args[2] ואת args[3] מ string ל boolean ואז מחשבים את ערך פעולת ה and (וגם) עליהם.

תוכנית ראשונה

```
public class Class1 {
   public static void main(String[] args){
      System.out.println(args[0]);
      System.out.println(1 + Integer.parseInt(args[1]));
      System.out.println(Boolean.parseBoolean(args[2]) &&
                                          Boolean.parseBoolean(args[3]));
   }
}
```

```
java Class1 abc 12 true false
```

```
abc
13
false
```

מחרוזות – על קצה המזלג

```
public class Class1Strings {
   public static void main(String[] args){
      System.out.println(args[0]);
      System.out.println(args[0].length());
      System.out.println(args[0].charAt(0));
      System.out.println(args[0].toUpperCase());
   }
}
```

```
java Class1Strings abc
```

```
abc
3
a
ABC
```

התכנית להיום

טעימה משפת Java

פונקציית main

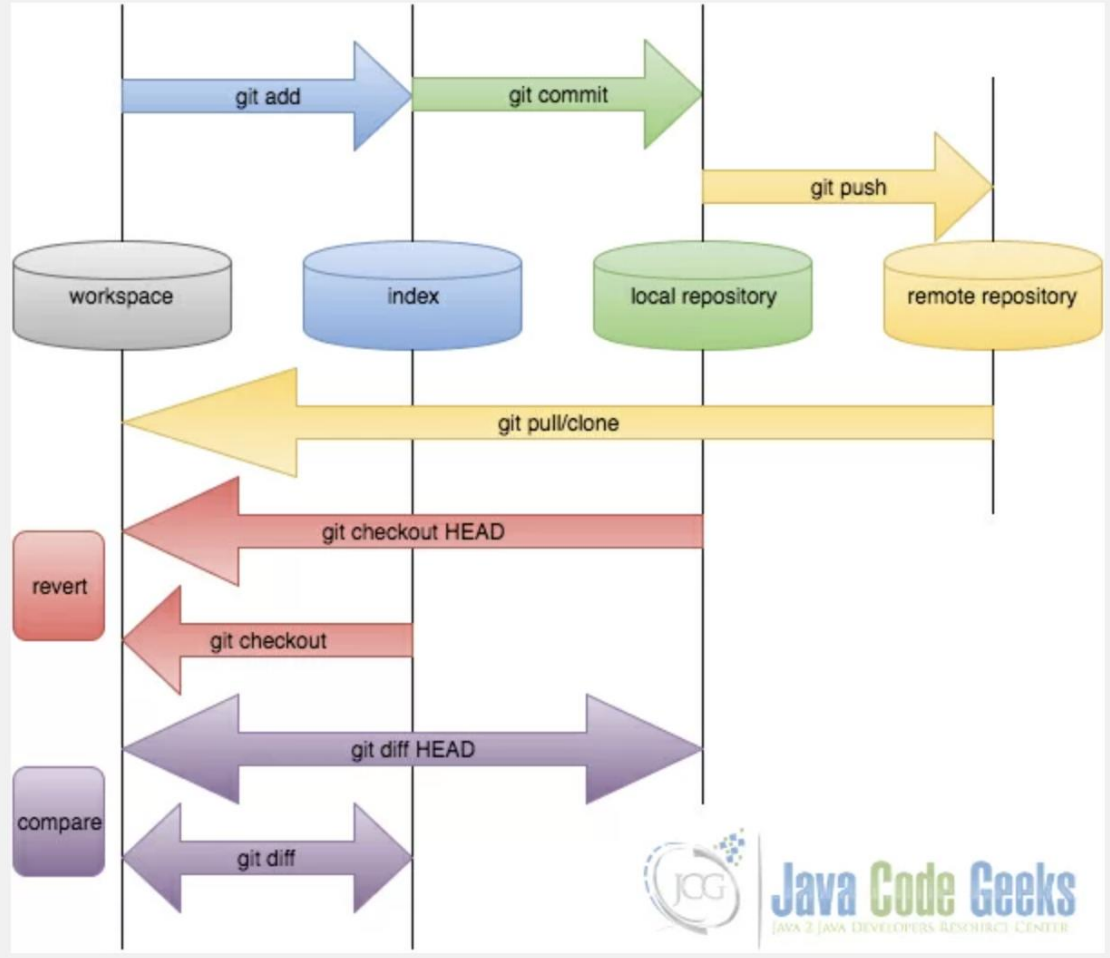
הכרות בסיסית עם git

תכנות בסיסי ב Java - בביתה ובעבודה עצמית 1:
8 הטיפוסים היסודיים, ביטויים ואופרטורים, טיפוסים שאינם יסודיים, טיפוס המחרוזת וטיפוס המערך

מה מאפשר לנו ה Git

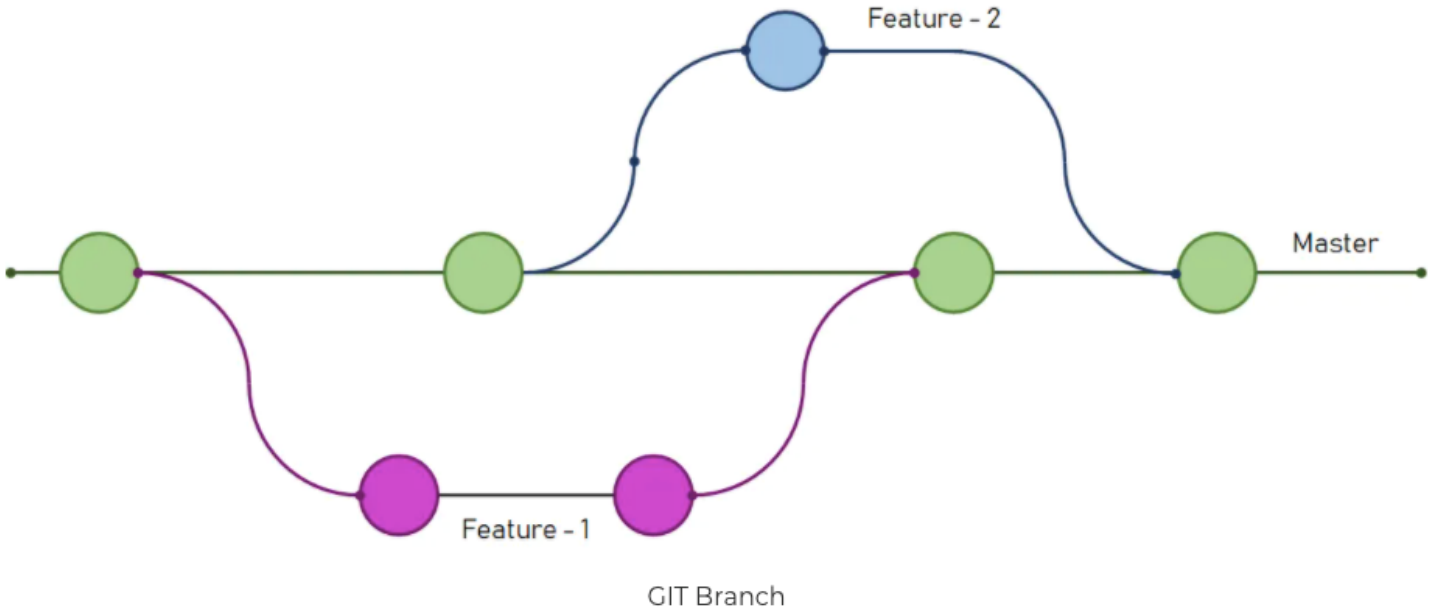
- ניהול גירסאות של הקוד:
  - מעקב אחרי שינויים.
  - חזרה אחורה.
  - גיבוי.
- עבודה בצוות:
  - עדכון של שינויים שבוצעו ע"י חברי צוות אחרים.
  - מיזוג (אוטומטי או ידני) במידה ושני חברי צוות משנים את אותו הקוד.
- ככה עובדים היום בכל מקום (ואם זה לא Git, זה כלי בקרת תצורה אחר עם מאפיינים דומים)

איך זה ניראה?



מתוך: https://examples.javacodegeeks.com/software-development/git/git-tutorial-beginners/

הסתעפויות



GIT Branch

From: https://digitalvarys.com/git-branch-and-its-operations/

התכנית להיום

טעימה משפת Java

פונקציית main

הכרות בסיסית עם git

תכנות בסיסי ב Java - בביתה ובעבודה עצמית 1:
8 הטיפוסים היסודיים, ביטויים ואופרטורים, טיפוסים שאינם יסודיים, טיפוס המחרוזת וטיפוס המערך

נושאים להמשך השיעור

- הערות
- 8 הטיפוסים היסודיים
- ביטויים ואופרטורים
- טיפוסים שאינם יסודיים
- טיפוס המחרוזת
- טיפוס המערך

הערות

- התוכנית מיועדת להיקרא על ידי המחשב (למעשה על ידי הקומפיילר), אבל גם על ידי תוכניתנים
- הערות הן טקסט בתוכנית שמיועד לקוראים אנושיים

```
/**
 * This is the first class I've ever written
 * @author Course Lecturer
 */
public class HelloWorld {

  /* This is the entry point of my application.
   * as you could see not so interesting...
   */
  public static void main(String[] arguments) {
    System.out.println("Hello World"); // prints "Hello World"
  }
}
```

סוגי הערות

- בג'אווה שלושה סוגי הערות:
- // הערה עד סוף השורה
- /* הערה רגילה, יכולה להתפרס על מספר שורות */
- /** הערת תיעוד (יכולה להתפרס על מספר שורות) */

- הערות לתיעוד שמופיעות לפני הגדרת מחלקה, שדה, או שירות עוברות, בעזרת כלי שנקרא javadoc לתיעוד המקוון של המחלקה

- הערות לתיעוד הן מובנות, ויש להן פורמט מיוחד שמיועד לאפשר לתוכניתן לתעד את הארגומנטים של שירות, את משמעות ערך החזרה, וכדומה

- כתבו הערות על מנת לבאר את הקוד:
  - הערות אודות המובן מאליו רק מכבידות: `i++ // add one to i`
  - אבל הערה טובה יכולה לחסוך הרבה זמן למי שקורא את הקוד

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

35

קשירות טיפוסים חזקה

- שלא כמו בשפת Python, ב- Java יש צורך בהגדרת טיפוס הנתונים של כל משתנה לפני השימוש בו. כמו כן, הגדרת חתימות למתודות מציינות את טיפוסי הנתונים שעליהן הם פועלות.

- מדוע?
  - **יעילות בזמן החישוב**
    - לדוגמא: פעולות חשבון על מספרים שלמים מהירות יותר מאשר פעולות על מספרים עשרוניים (בייצוג נקודה צפה)
  - **חסכון בהקצאת זיכרון**
    - לדוגמא: לייצוג ציון במבחן נדרשות פחות ספרות מאשר לייצוג מספר תעודת הזהות
  - **אופי הנתונים מגדיר פעולות שניתן לבצע עליהם**
    - לדוגמא: שנת לידה היא נתון מספרי שניתן לחסר לצורך חישוב גיל. שם פרטי יהיה נתון מטיפוס מחרוזת, שעליו אין הגיון לבצע חיסור

הגדרת טיפוס לכל נתון מאפשרת לזהות שגיאות בשלב הקומפילציה של התוכנית

36

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`**

- בזיכרון המחשב נשמר המידע בפורמט בינארי
- **סיבית** (bit) היא ספרה בינארית ('0' או '1')
- **בייט** (octet, byte, ברבים: "בתים") הוא קבוצה של 8 סיביות

- לפני שנדון בטיפוסים השונים, נתבונן בשימוש במשתנה מטיפוס **`int`**
- **`int`** הוא מספר שלם חיובי או שלילי המיוצג בזיכרון ע"י 4 בתים (32 ביט) בבסיס 2

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
- בעת, הגישה למשתנה i תקינה ויודפס למסך: i=5

אתחול משתנה מקומי

- ניתן לשלב את ההצהרה וההשמה ע"י משפט אתחול:

```
int i = 7;
```

- בדוגמא זו המשתנה `i` נוצר (הוקצה לו זיכרון) והושם לו ערך באותה הפעולה
- כך מובטח כי הגישה ל- `i` תהיה תמיד בטוחה
- אין ב Java אתחול ברירת מחדל למשתנים מקומיים

השמה

- תחביר ההשמה הוא:

<variable> = <expression>

השמה מתבצעת "מימין לשמאל":

1. מחושבת תוצאת הביטוי שבאגף ימין
2. ערך הביטוי נכתב לתוך המשתנה שבאגף שמאל

- לדוגמא: בהשמה `i = 1 + 2` מושם הערך 3 לתוך `i`

- זהו הבסיס לתכנות אימפרטיבי (או פרוצדורלי) – תהליך החישוב מתקדם ע"י שינוי ערכי משתנים

- תכנות מונחה עצמים ב Java נבנה על התשתית האימפרטיבית

השמה (המשך)

- מה קורה אם הביטוי באגף ימין הוא משתנה בעצמו?

```
int x = 5;
int y = x;
```

- מה יקרה ל- `y` אם נשנה עבשיו את `x`?

```
x = 3;
```

- מה יקרה ל- `x` אם נשנה עבשיו את `y`?

```
y = 7;
```

בתהליך ההשמה מועתקת תוצאת חישוב הביטוי שבאגף ימין.
בשונה ממזיגת נוזל ממיכל למיכל

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

טיפוסים שלמים

- Java מספקת ארבעה סוגי טיפוסי משתנים שלמים:
  - **`byte`** - 8 סיביות בייצוג משלים 2
  - **`short`** - 16 סיביות בייצוג משלים 2
  - **`int`** - 32 סיביות בייצוג משלים 2
  - **`long`** - 64 סיביות בייצוג משלים 2

- משתנים מטיפוס שלם יכולים לייצג מספרים שלמים:
  - חיוביים, שליליים או אפס

- הטווח של כל טיפוס נקבע על פי מספר הסיביות בייצוג
- אין ב- Java אפשרות לייצג מספרים אי-שליליים בלבד, כדוגמת טיפוס **unsigned int** בשפת C

טיפוסי נקודה צפה

- ייצוג ערכים ממשיים מתבצע ב Java ע"י הטיפוסים:
  - `float` – 32 סיביות – 1 לסימן, 8 לחזקה (של 2), 23 לשבר
  - `double` – 64 סיביות - 1 לסימן, 11 לחזקה, 52 לשבר

- הייצוג הפנימי שלהם מספרים ממשיים שונה מהייצוג של מספרים שלמים, והוא מבוסס על תקן בשם IEEE-754
- פעולות על מספרים בייצוג נקודה צפה איטיות יותר מפעולות על מספרים שלמים

תווים וסימנים

- ג'אווה מספקת טיפוס פרימיטיבי לייצוג תווים: **char**
- תווים הם הסמלים שאנו משתמשים בהם לייצוג טקסט, והם כוללים אותיות (של כל השפות), ספרות, סימני פיסוק ועוד

```
char c = '?';
System.out.println("A question mark is " + c);
```

- בג'אווה תווים מיוצגים על ידי ערכים מספריים לפי קידוד Unicode (16 סיביות)
- ניתן לייצג בתוכנית קבועים מטיפוס char ע"י ציון התו בין גרשיים או באמצעות ציון הערך המספרי שלו

```
char c = 63;
System.out.println("A question mark is " + c);
```

תווים וסימנים

- בפעולות על תווים או מחרוזות, השפה מתייחסת לתווים כתווים, ופועלת בהתאם.
- למשל, שרשור תו למחרוזת משרשר אותו כתו, לעומת שרשור של שלם, שמשרשר למחרוזת את הייצוג העשרוני של הערך:

```
char c = '?';
String str = "The letter "+c; // "The character ?"

int i = 63;
String t = "The number "+i; // "The number 63"
```

הטיפוס הבוליאני

- משתנים בוליאניים (**boolean**) יכולים לקבל שני ערכים: false ו- true
- להבדיל מטיפוס **char**, אין לנו כמתכנתים מידע על ייצוג הפנימי של טיפוסים בולאניים ולא ניתן להתייחס אליהם כשלמים (0 או 1)

☑ `boolean z = false;`

☒ `boolean q = 0;`

- אופרטורים של השוואה (בין מספרים), מחזירים ערך בוליאני.
- לדוגמה: ==, =!, <, < , =>, >=

```
boolean z;
z = 4>3;
System.out.println("z=" + z); // z=true
```

קבועים (literals)

- Java משייכת לכל ערך בתוכנית טיפוס, כדי לדעת לאיזה משתנה ניתן יהיה להשים אותו בעתיד

- קבועים הם ערכים שמופיעים ישירות בקוד המקור בג'אווה

- הם כוללים מספרים שלמים, מספרים בנקודה צפה, תווים בתוך ציטוט בודד, מחרוזות תווים בתוך ציטוט כפול, והמילים השמורות true, false, null

- לדוגמא:
  - 3 נחשב כ- **`int`**
  - ‘3’ נחשב כ- **`char`**
  - ”3“ נחשב כ- **`String`**
  - 3.0 נחשב כ- **`double`**
  - **true** נחשב כ- **`boolean`**

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

- אנלוגיה: האם **בטוח** לשפוך דלי שקיבולתו 8 ליטר לדלי שקיבולתו 4 ליטר ?
- לא *בטוח*, אף על פי שלפעמים זה יצליח, למשל אם היו בדלי המקורי רק 2 ליטר

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;
```

המרה מפורשת של קבועים

- מה לא בסדר בשורה הבאה?

```
float f = 3.0;
```

- הליטרל 3.0 מתפרש ע"י המהדר כ double – המרה הכרחית

- עבור ליטרלים קיים תחביר המרה מקוצר – הוספת האות f (או F) מיד לאחר המספר:

☑ `float f = (float)3.0`

שקול ל:

☑ `float f = 3.0F`

שמות (מזהים, identifiers)

- מזהה הוא שם שניתן למרכיב בלשהו של תכנית, כגון מחלקה, שרות, משתנה
- מזהה יכול להיות באורך בלשהו, ולהכיל אותיות, ספרות ואת הסימנים $ ו- _ (וכן סימנים נוספים שלא נפרט)
- מזהה אינו יכול *להתחיל* בספרה
- מומלץ להשתמש בשמות משמעותיים

- דוגמאות:

```
int examGrade = 92;
double PI = 3.1415927;
float salary = income * (1 - incomeTaxRate);
```

מבנה לקסיקלי

- תכנית היא סידרה של תווים, הנחלקים ליחידות בסיסיות הנקראות אסימונים (tokens) בגון מספרים, מזהים וכו'

- ג'אווה היא case sensitive בלומר עושה אבחנה בין אות קטנה לאות גדולה
  - לדוגמא המזהה grades שונה מהמזהה Grades

- ג'אווה מתעלמת מ"רווחים לבנים" (רווחים, סימני טאב, שורה חדשה וכו') פרט לאלה שמופיעים בתוך תווים מצוטטים ומחרוזות ליטרליות.
  - למשל: **"astring"** שונה מ **"a string"**

משפטים וביטויים

- *משפט* (statement) **מבצע פעולה**
  - בגון: משפט השמה, משפט תנאי, משפט לולאה, קריאה לפונקציה שאינה מחזירה ערך (void)

- *ביטוי* (expression) הוא מבנה תחבירי **שניתן לחשב את ערכו**
  - בגון: הפעלת אופרטור (בשקף הבא), קריאה לפונקציה המחזירה ערך, משפט השמה

- פונקציות ב-Java (מתודות) הן סדרה של משפטים
- *משפט* **מבצע פעולה** על ביטויים (expression)
- ההפרדה אינה מלאה - ישנם משפטים אשר ניתן לחשב את ערכם (כגון משפט השמה, או אופרטור הקידום)

סימני פיסוק

- סימני פיסוק מופיעים גם הם כאסימונים משני סוגים:
  - מפרידים:

```
( ) { } [ ] < > : ; , . @
```

  - אופרטורים:

```
+ - * / % & | ^ << >> >>>
+= -= *= /= %= &= |= ^= <<= >>= >>>=
= == != < <= > >=
! ~ && || ++ -- ?
```

- נראה בהמשך את משמעות האופרטורים, אבל לא את כולם

ביטויים ואופרטורים

- ביטויים (אריתמטיים או אחרים) מוגדרים באופן הבא:
  - קבוע (literal) - הוא ביטוי שמייצג את ערכו
  - משתנה הוא ביטוי שערכו כערך שיש כרגע למשתנה
  - הפעלה של אופרטור על ביטוי (או ביטויים) מתאימים היא ביטוי

- רוב האופרטורים (לא כולם) נכתבים בכתיב infix
  - כמו x + 1,

- כל אופרטור קובע את מספר הארגומנטים שלו, את הטיפוסים שלהם, ואת הטיפוס של הערך המוחזר

- לכל אופרטור סדר קדימות, וכן אסוציאטיביות (לימין או לשמאל); סוגריים מאפשרים לשלוט על סדר הפעולות

אופרטורים ביניריים לפי סדר הקדימות שלהם (טבלה חלקית)

כפל, חילוק, שארית (גם לנקודה צפה)	* / %
חיבור (ושרשור מחרוזות, גם למספר, תו), חיסור	+ -
הזזה של סיביות שמאלה, ימינה אריתמטי ולוגי	<< >> >>>
גדול מ, קטן מ, גדול או שווה, קטן או שווה	< > <= >=
שוויון ואי שוויון	== !=
AND (לערכים בוליאניים או שלמים כווקטורי סיביות)	&
XOR (בנ"ל)	^
OR (בנ"ל)	|
Short-circuit AND (מתעלם מהאופרנד השני אם הראשון קובע את התוצאה)	&&
Short-circuit OR (בנ"ל)	||

אופרטורים בינריים

- סדר הקדימות נועד לצמצם את הצורך בשימוש בסוגריים

```
int result = 4 * argument + 5;
```

שקול ל:

```
int result = ((4 * argument) + 5);
```

```
boolean isLegalGrade = grade >= 0 && grade <= 100;
```

שקול ל:

```
boolean isLegalGrade = ((grade >= 0) && (grade <= 100));
```

- הערך המוחזר של אופרטור ההשמה הוא הערך שהושם בפועל, כך שניתן לשרשר השמות:

```
int i = j = k = 0;
```

שקול ל:

```
int i = (j = (k = 0));
```

קידום (prefix)

- הוספה והורדה של 1 כ"כ שכיחים עד שהומצא אופרטור מיוחד לכך:

```
x += 1
```

שקול ל

```
++x
```

- לדוגמה:

```
int x = 5;
++x;
System.out.println(x);      // מה יודפס ?
```

- מה יודפס אם נשלב את שתי השורות האחרונות בדוגמה:

```
int x = 5;
System.out.println(++x);      // מה יודפס ?
```

- הערך המוחזר של הפעולה הוא הערך החדש של x

קידום (postfix)

- ב Java קיים אופרטור קידום נוסף `x++`
- ההבדל בינו ובין `x++` הוא שהערך המוחזר של הפעולה הוא הערך המקורי של `x`

```
int x = 5;
System.out.println(x++);
System.out.println(x);
```

```
// 5 יודפס מה
// 6 יודפס מה
```

- בדומה קיימים האופרטורים `x--` ו- `--x`
- מומלץ שלא להשתמש באופרטורי הקידום וההפחתה כביטויים אלא רק במשפטים (פעולה בפני עצמה)

אופרטורים אונאריים

מחזיר את x ומקדם/מוריד אותו ב- 1	`x++` `x--`
מקדם/מוריד ב- 1 ואז מחזיר את הערך החדש	`++x` `--x`
מספר נגדי (הפיכת סימן)	`-`
הפיכת כל הסיביות של מספר שלם	`~`
הפיכה של ערך בוליאני	`!`

- האופרטורים האונאריים קודמים לבינאריים
- אופרטורים בינאריים אסוציאטיביים לשמאל, בעוד שאופרטורים אונאריים ואופרטור השמה אסוציאטיביים לימין
  - בלומר: `i = j = k = 0` שקול ל: `i = (j = (k = 0))`
  - אבל: `i + j + k` שקול ל: `((i + j) + k)`

משתנים שאינם יסודיים
(non primitive variables)

- פרט ל-8 הטיפוסים שסקרנו עד כה, כל שאר הטיפוסים ב Java אינם פרימיטיביים

- הספריה התקנית של Java מכילה אלפי טיפוסים (!) ואנו כמתכנתים נשתמש בהם ואף ניצור טיפוסים חדשים

- 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

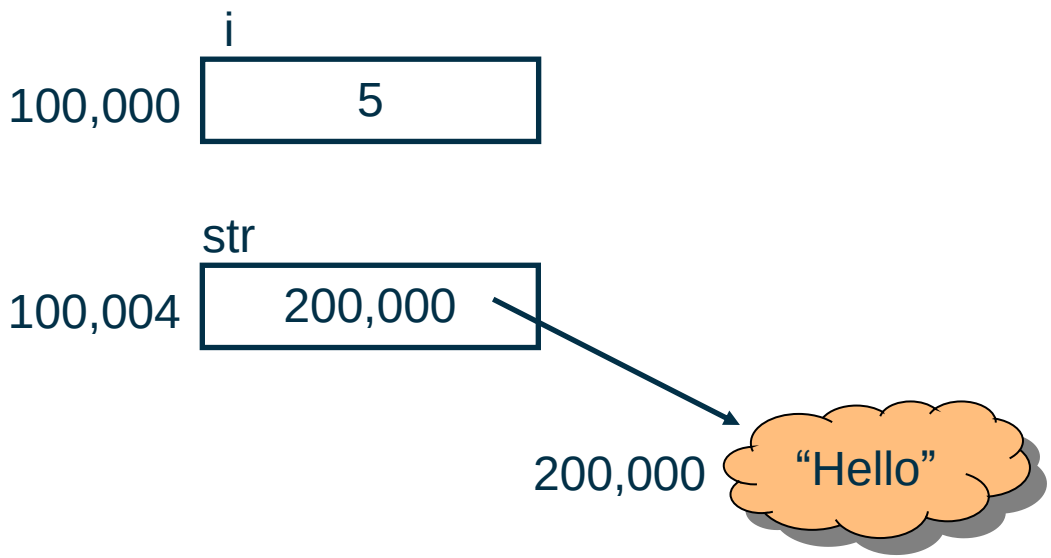
- מערכים ומחרוזות אינם טיפוסים יסודיים, אולם מכיוון שאנו שנזדקק להם כבר בשיעורים הקרובים נדון בקצרה בטיפוסי הפנייה

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

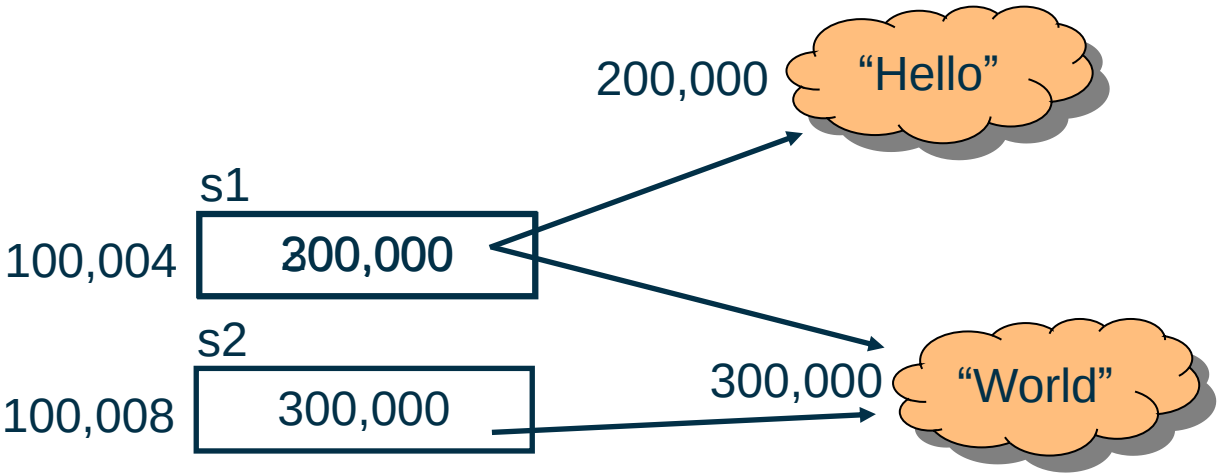
פעולות על הפניות

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

➡ String s1 = “Hello”;

➡ String s2 = “World”;

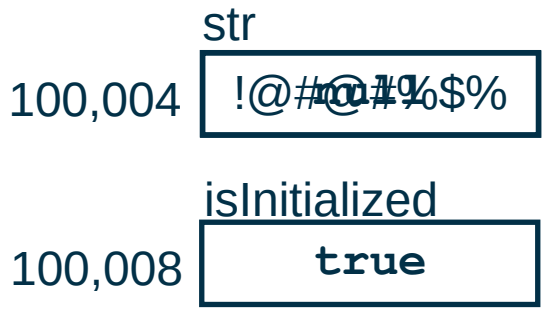
➡ s1= s2;



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

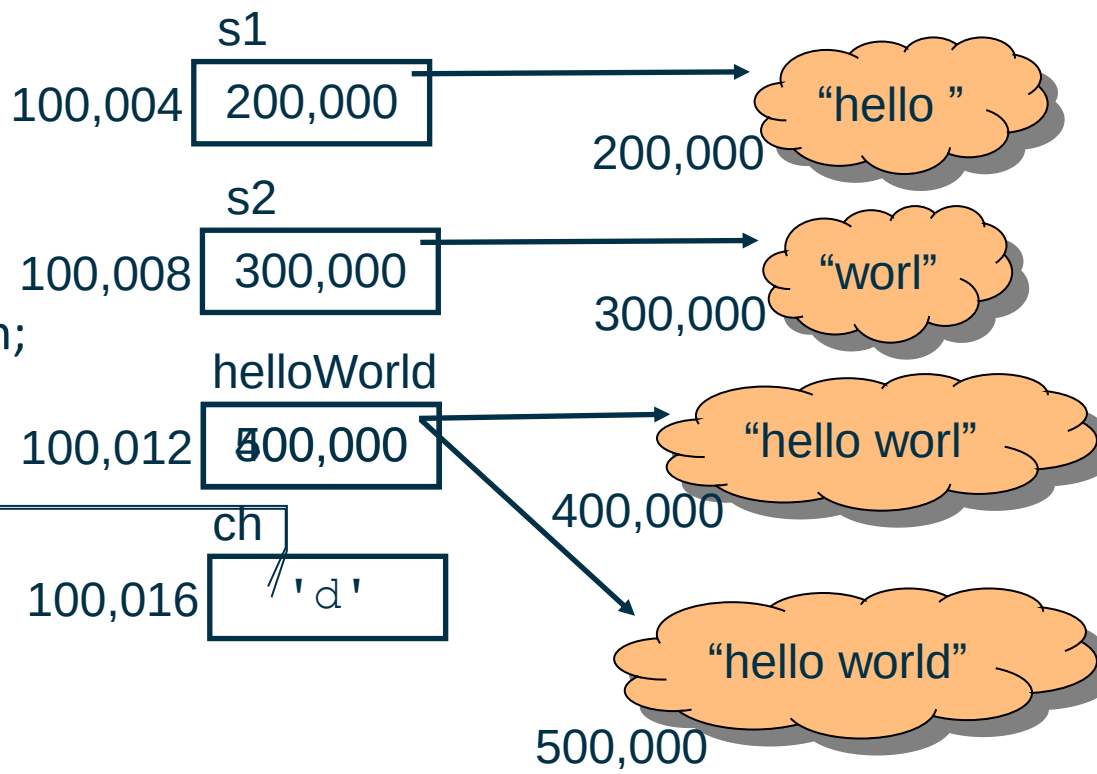
```
String str;
str = null;
boolean isInitialized = (str == null);
```



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;



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

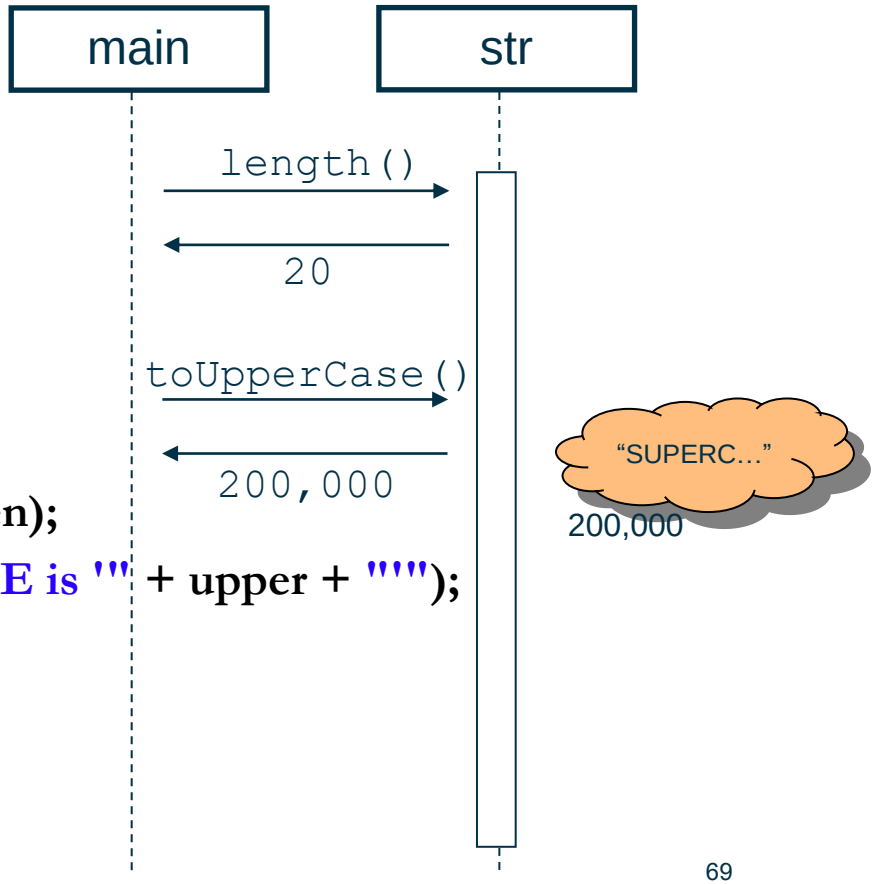
- הבקשות השאלות והמאפיינים הפנימיים משתנים מעצם לעצם לפי טיפוסו (אם כי ישנן בקשות שאפשר לבקש מכל עצם ב Java)

דוגמה

- בדוגמה 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 טיפוס (בין אם פרימיטיבי או הפניה)
- למשל מערך של איברים מטיפוס int:

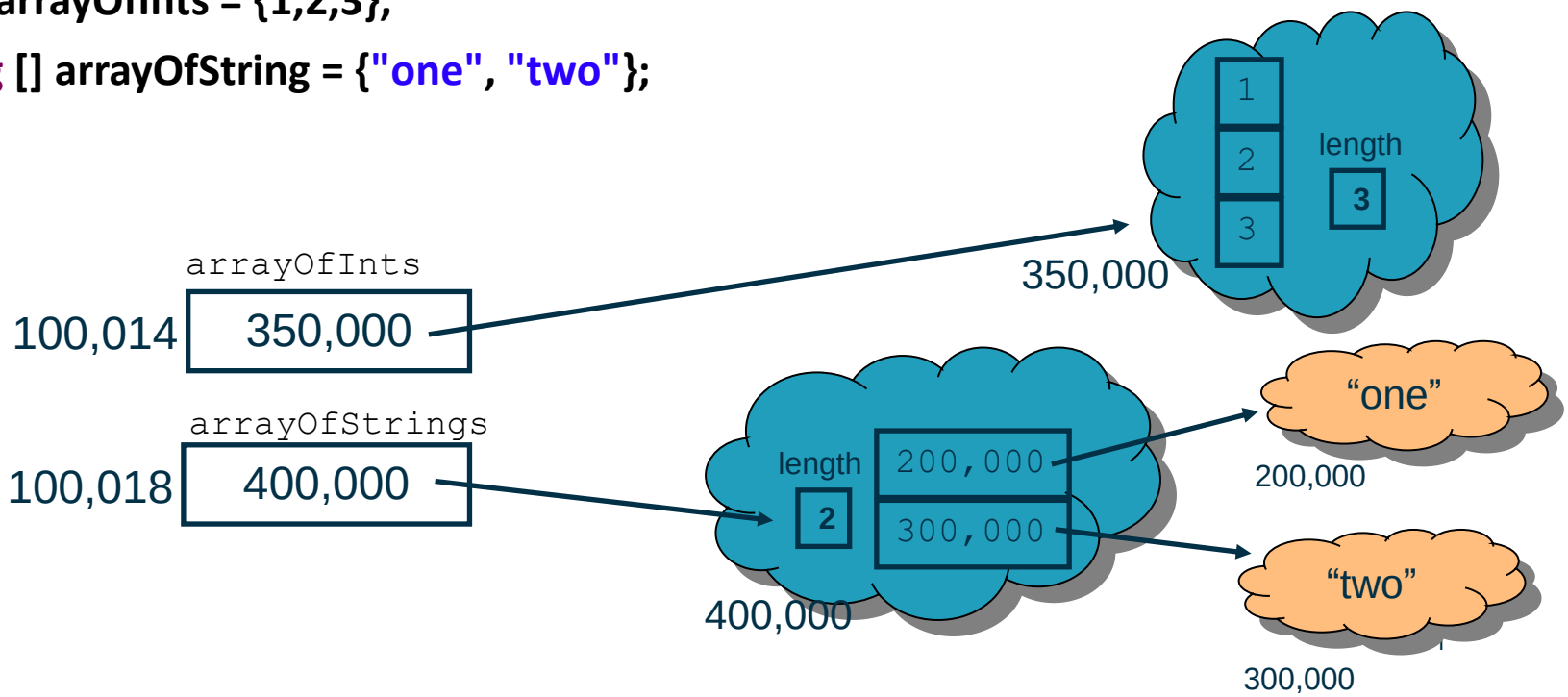
2	3	5	7	11	13	17	19	23	29

- 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"};
```



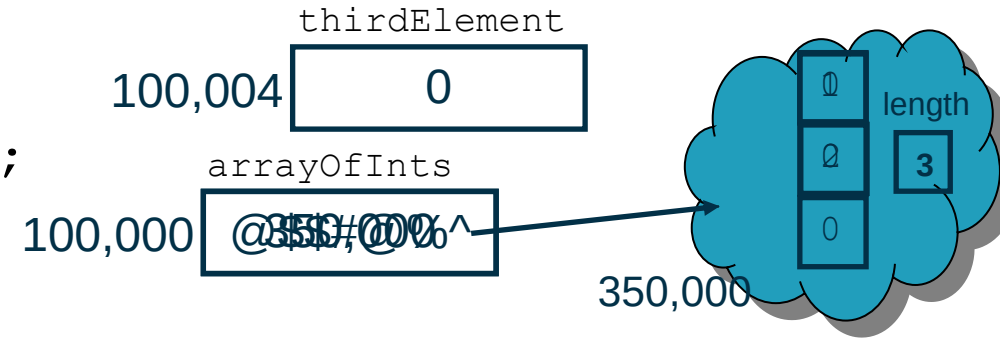
מערכים

- 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

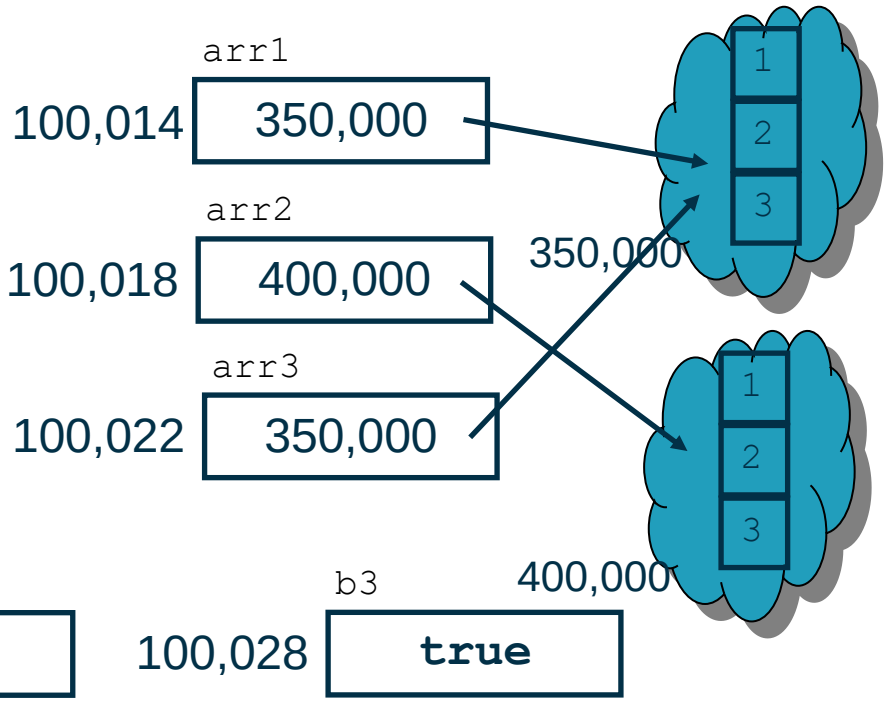
- אורכו של מערך, הוא מאפיין פנימי אשר ניתן לגשת אליו ישירות בעזרת אופרטור הנקודה

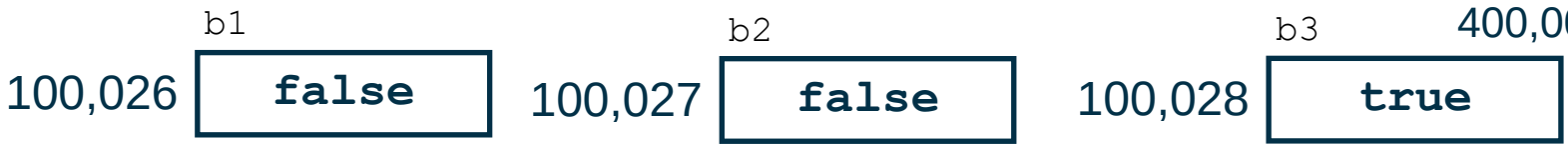
```
int [] arrayOfInts = {1,2,3};
System.out.println("The size of my array is " +
 arrayOfInts.length);
```

הפניות ואופרטור ההשוואה (==)

- כאשר אופרטור ההשוואה (==) מופעל על משתני הפניה, הוא 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);
```

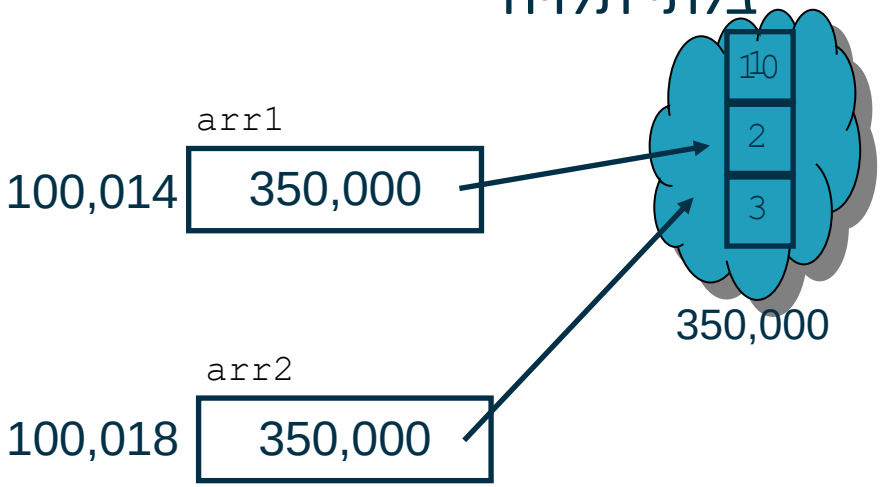




שיתוף (sharing, aliasing)

- 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 (+)