תוכנה 1 בשפת Java

שיעור מספר 1: "שלום עולם"

ליאור וולף

מתי שמרת

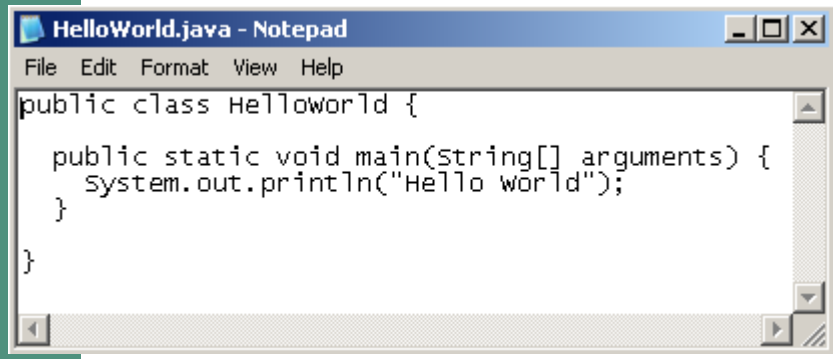
בית הספר למדעי המחשב

אוניברסיטת תל אביב

מה בתכנית?

- טעימה משפת Java
- פונקציית `main`
- 8 הטיפוסים היסודיים
- ביטויים ואופרטורים
- טיפוס המחרוזת וטיפוס המערך

שלום עולם



HelloWorld.java

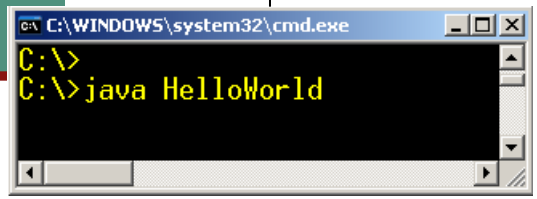
compile

HelloWorld.class

Run on Windows

Run on Solaris

Run on Mac



Write Once – Run Anywhere !

המפרש (interpreter)

- את הקוד שנכתב בשפת Java מריץ *מפרש*
  - בדומה לשפת Scheme
- 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 כגון 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?

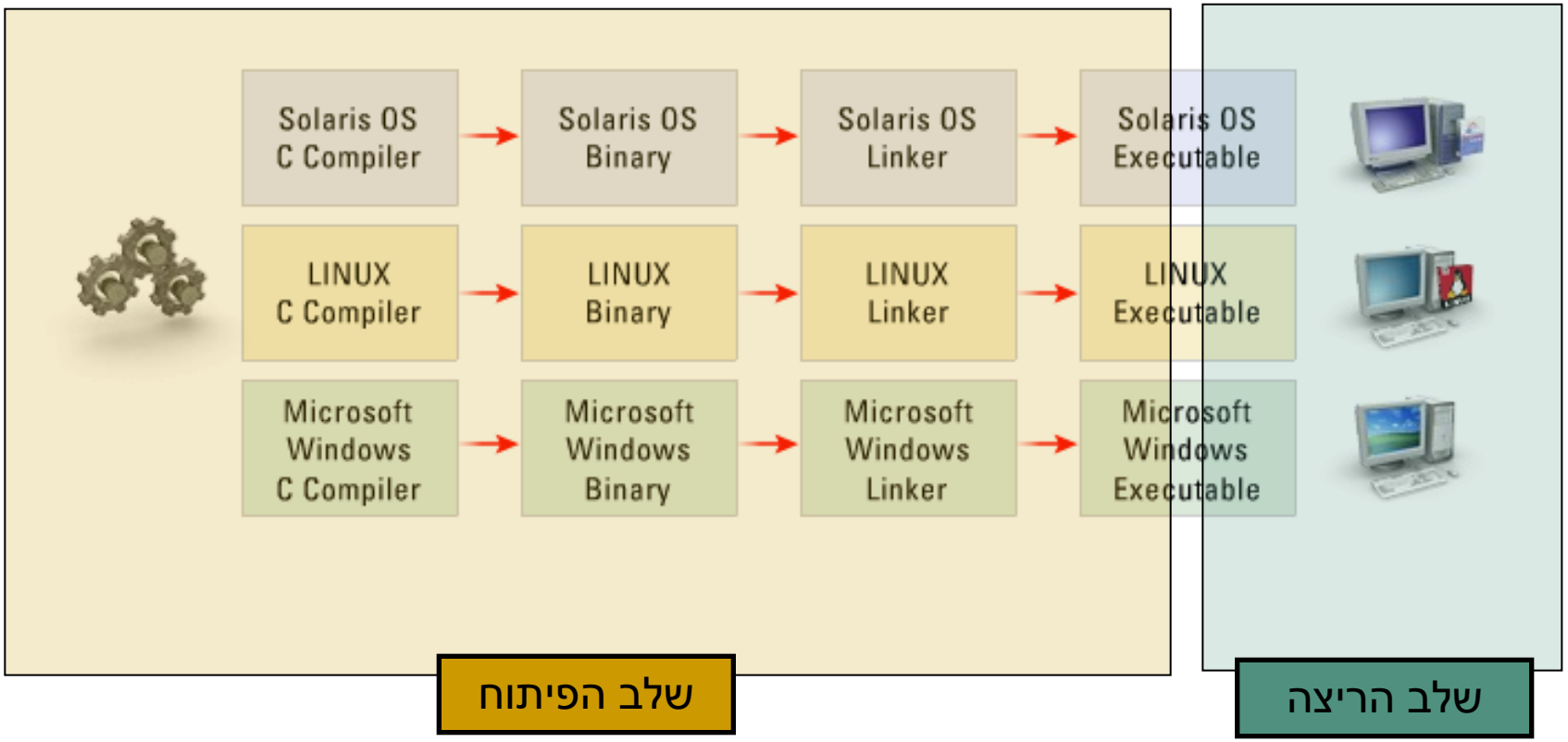
- זאת מכיוון שאיננו יודעים מראש על איזה מחשב בדיוק תרוץ תוכנית ה- Java שכתבנו
- תוכניות Java *חוצות סביבות* (cross platform)
  - סביבה = חומרה + מערכת הפעלה
  - תוכנית שנכתבה והודרה במחשב מסוים, תוכל לרוץ בכל מחשב אשר מותקן בו מפרש ל- Java

המכונה המדומה (Java Virtual Machine)

- 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

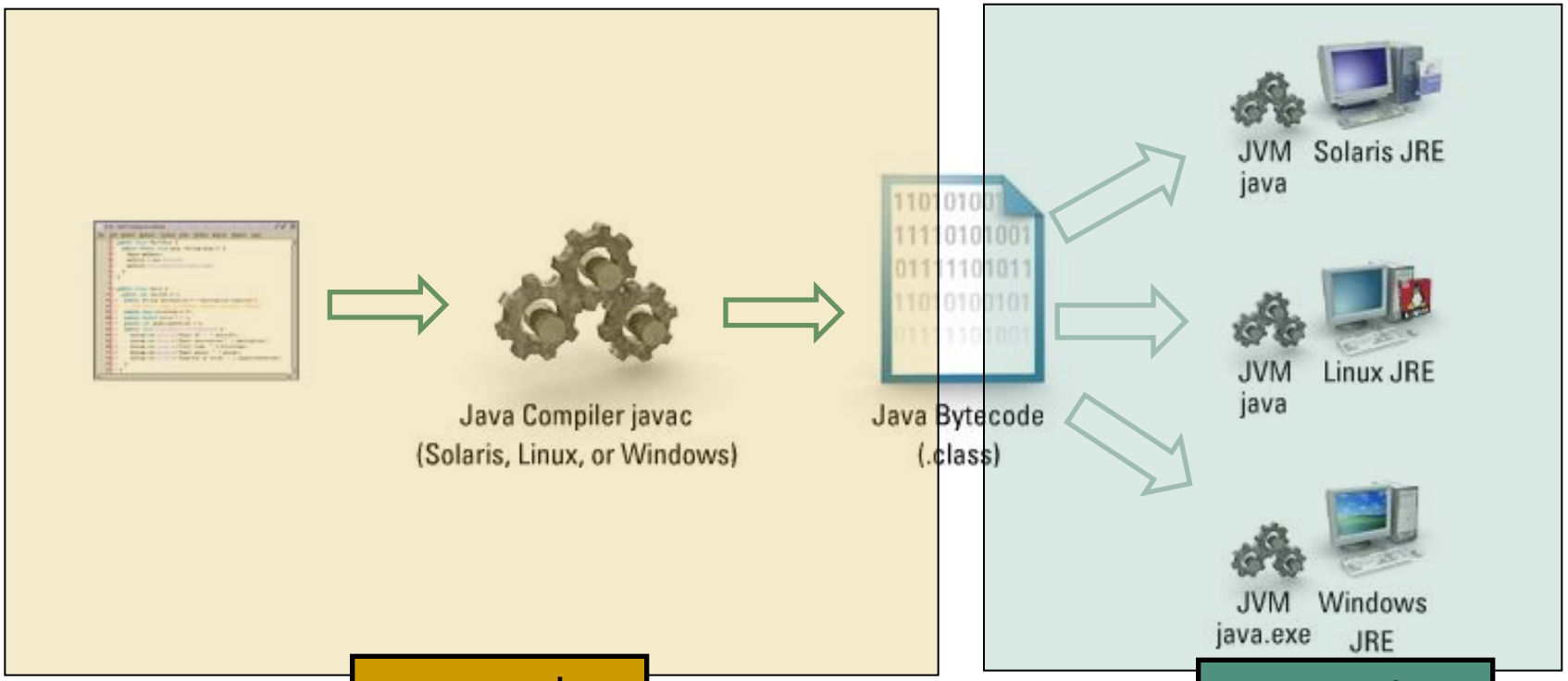
תלות בסביבה (platform specific)

- בשפות אחרות (C/C++) אין הדבר כך:



עצמאות סביבתית (platform independence)

- ב Java תכונה זו אפשרית הודות לרעיון "שפת הביניים" וה JVM הנפרד לכל סביבה



שלב הפיתוח

שלב הריצה

שלום עולם

המחלקה ציבורית – ניתן להשתמש בה ללא הרשאות מיוחדות

הגדרת מחלקה בשם HelloWorld. בשלב זה, נזהה מחלקה עם קובץ באותו שם

```
public class HelloWorld {

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

}
```

יצירת תחום (scope)

חתימת המתודה

הגדרת מתודה (פונקציה)

גוף המתודה

המתודה main

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

- כאשר אנו מריצים מחלקה ה JVM מחפש מתודה עם חתימה זו, ומריץ אותה
  - 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 (void = ריק)

המתודה main

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

- הגדרת פרמטר String[] arguments למתודה *בשם* arguments *ומטיפוס* מערך של מחרוזות
- לכל המשתנים ב Java יש טיפוס המעיד על סוג וטווח הערכים שיכולים להיות מאוחסנים במשתנה (למשל: מספר שלם, תו, משפט, מערך ואחרים)
- שם המשתנה אינו חלק מהחתימה של המתודה

המתודה main

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

- System.out.println קריאה למתודה (method call, זימון מתודה) – אנו משתמשים כאן בשמה המלא של המתודה (qualified name) המכיל את התו '.' (נקודה)
- העברת ארגומנט מטיפוס מחרוזת (String) – משפט עטוף במרכאות הוא מטיפוס מחרוזת
- משפטים ב Java מסתיימים בתו ';' (נקודה-פסיק)

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
- בשלב ראשון בקורס נדון רק בטיפוסים היסודיים

קשירות טיפוסים חזקה

- שלא כמו בשפת Scheme, 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

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`

- בזכרון 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 בזכרון 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 – סימן ה'=' ב- Java אינו מציין השוואה אלא השמה (בדומה ל set! בשפת Scheme) – הערך 5 ייכתב לתוך הזיכרון שהוקצה למשתנה i
- כעת, 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 זכרון) 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 עכשיו את x?

```
x = 3;
```

- מה יקרה ל- x אם נשנה עכשיו את y?

```
y = 7;
```

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

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
- למשל, משתנה מטיפוס **`byte`** יכול להכיל מספרים מ- 127 עד 128-
- אין ב- Java אפשרות לייצוג 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
- פרטים נוספים על הייצוג הפנימי יילמדו בקורס "פרויקט תוכנה" ובקורס "ארכיטקטורת מחשבים"

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);
```

- כדי לייצג את התווים בזיכרון המחשב מקובל להעזר בטבלה הנותנת לכל תו מספר סידורי (אי שלילי). טבלה זו נקראת טבלת קידוד (encoding character)
- בג'אווה תווים מיוצגים על ידי קידוד Unicode (16 סיביות)
  - טבלה זו גדולה מספיק כדי להכיל את רוב מערכות ה- א"ב הקיימות (להבדיל מטבלת ה ASCII שהתבררה כקטנה מדי)
  - למשל, התו 'A' מקודד על ידי המספר 65 , ואילו התו 'א' מקודד על ידי המספר 1488

תווים וסימנים

- ניתן לייצג בתוכנית קבועים מטיפוס char ע"י ציון התו בין גרשיים או באמצעות ציון הערך המספרי שלו (למשל בעזרת: http://unicode.coeurlumiere.com):

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

- מומלץ להכיר כמה תווים אשר אין להם ייצוג כסימן על המסך, ולכן ניתן להם ייצוג מיוחד:
  - שורה חדשה: '\n'
  - טאב: '\t'

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"
```

- פרטים נוספים על שימוש בתו כמספר (לא מומלץ!) בקובץ הערות שנמצא באתר

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:
  - == (שוויון),
  - =! (אי שוויון),
  - <, > , =<, => (קטן מ, גדול מ, קטן או שווה, גדול או שווה)

```
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`

- תחביר דומה קיים עבור טיפוסים נוספים – פרטים בתרגול ובדוגמאות שבאתר הקורס

שמות (מזהים, identifiers)

- מזהה הוא שם שניתן למרכיב כלשהו של תכנית, כגון מחלקה, שרות, משתנה
- מזהה יכול להיות באורך כלשהו, ולהכיל אותיות, ספרות ואת הסימנים $ ו- _ (וכן סימנים נוספים שלא נפרט)
- מזהה אינו יכול *להתחיל* בספרה
- בשונה מהכללים לגבי מזהים ב scheme, אך דומה לרוב השפות האחרות
- מומלץ להשתמש בשמות משמעותיים
- קיימות מוסכמות לגבי סוגי שמות (באתר הקורס)

- דוגמאות:

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

מילות מפתח בג'אווה (keywords)

- המילים במסגרת הן מילות מפתח בג'אווה
- הן מילים שמורות: אין להשתמש בהן כשמות בתכניות
- בנוסף, המילים true, false, null אינן מילות מפתח אבל גם הן שמורות ואין להשתמש בהן כמזהים

חסרות מימוש

abstract	continue	for	new	switch
assert	default	goto	package	synchronized
boolean	do	if	private	this
break	double	implements	protected	throw
byte	else	import	public	throws
case	enum	instanceof	return	transient
catch	extends	int	short	try
char	final	interface	static	void
class	finally	long	strictfp	volatile
const	float	native	super	while

מבנה לקסיקלי

- תכנית היא סידרה של תווים, הנחלקים ליחידות בסיסיות הנקראות אסימונים (tokens) כגון מספרים, מזהים וכו'
- ג'אווה היא case sensitive כלומר 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**
  - כגון: משפט השמה, משפט תנאי, משפט לולאה, קריאה לפונקציה שאינה מחזירה ערך (void)
- *ביטוי* (expression) הוא מבנה תחבירי **שניתן לחשב את ערכו**
  - כגון: 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 בינריים 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 (כנ"ל)	^
OR (כנ"ל)	|
Short-circuit AND (מתעלם מהאופרנד השני אם הראשון קובע את התוצאה)	&&
Short-circuit OR (כנ"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));
```

שינוי ערכו של משתנה

- השמה דורסת את ערכו הישן של משתנה. מה נעשה אם נרצה שהערך החדש יהיה תלוי בערך הנוכחי?
- לדוגמא:

```
int x = …
```

- איך נגדיל את x ב-5 ?
- **אפשרות א'** - ע"י שימוש במשתנה עזר:

```
int temp = x;
x = temp + 5;
```

- **אפשרות ב'** - ע"י שימוש ב-x עצמו:

```
x = x + 5;
```

- מדוע זה עובד? מכיוון שהשמות מתבצעות מימין לשמאל: קודם משוערך הביטוי שבאגף ימין ורק אח"כ הוא עובר השמה
- פעולה זו היא כה שכיחה עד שהומצא לה סוכר תחבירי משלה

השמה עם פעולה

- ג'אווה תומכת בסימון מקוצר עבור אופרטורים בינריים והשמה של התוצאה חזרה לתוך האופרנד הראשון

```
x += y;
```

שקול ל-

```
x = x + y;
```

- כמעט בכל האופרטורים הבינריים ניתן להשתמש כך

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

- השילובים הללו מופיעים אחרונים בסדר הקדימות, יחד עם אופרטור ההשמה הרגיל (=), כך שקודם צד ימין של הביטוי (y) מחושב, אחר כך מתבצעת הפעולה בין צד שמאל (x) ובין תוצאת החישוב, ואחר כך ההשמה

- ההשמה אסוציאטיבית לימין דבר המאפשר השמה מרובה (אפשרי אבל לא מומלץ):

```
x = y = <exp>
```

קידום (prefix)

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

```
x += 1
```

שקול ל-

```
++x
```

- לדוגמא:

```
int x = 5;
++x;
System.out.println(x); // מה יודפס 6
```

- מה יודפס אם נשלב את שתי השורות האחרונות בדוגמא:

```
int x = 5;
System.out.println(++x); // מה יודפס 6
```

- הערך המוחזר של הפעולה הוא הערך <u>החדש</u> של x

קידום (postfix)

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

```
int x = 5;
System.out.println(x++); // מה יודפס 5
System.out.println(x);   // מה יודפס 6
```

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

אופרטורים אונריים

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 האונריים קודמים לבינאריים
- אופרטורים בינאריים אסוציאטיביים לשמאל, אונריים והשמה לימין
  - כלומר: `i = j = k = 0` שקול ל: `i = (j = (k = 0))`
  - אבל: `i + j + k` שקול ל: `((i + j) + k)`

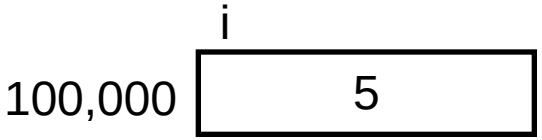
משתנים שאינם יסודיים (non primitive variables)

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

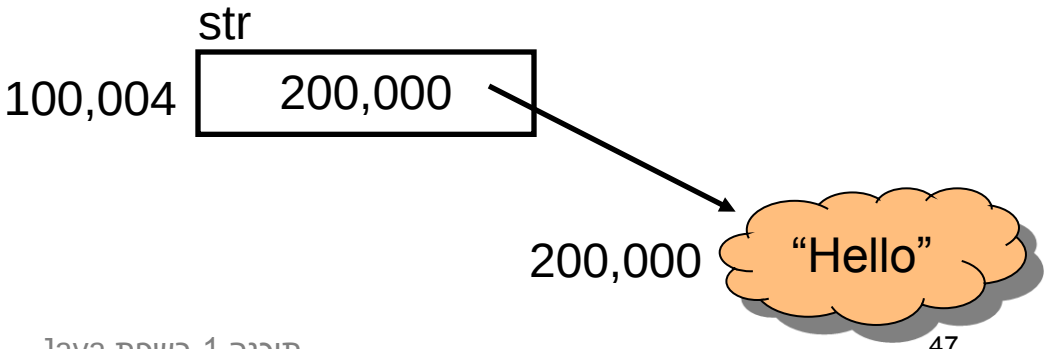
הפניות ומשתנים יסודיים

- 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

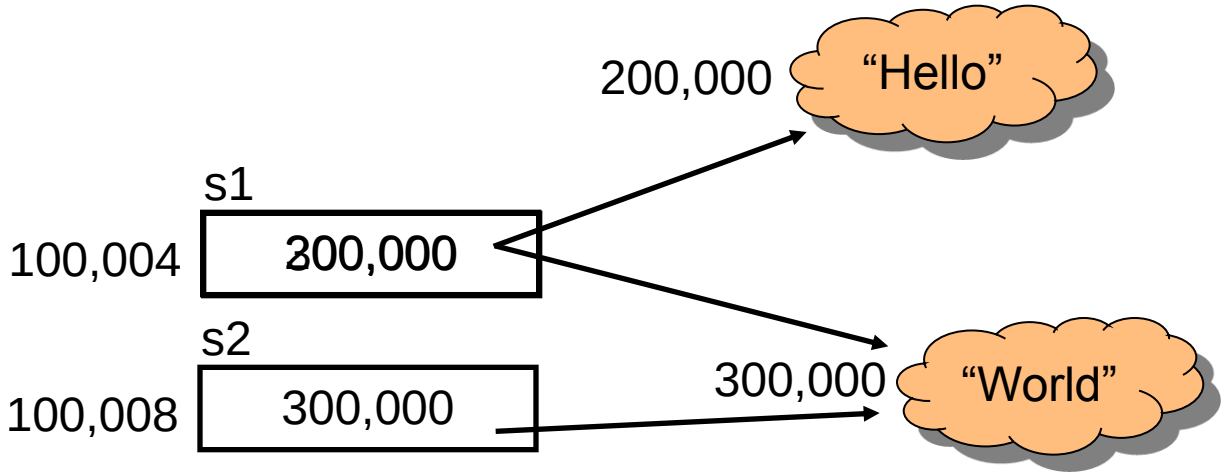
פעולות על הפניות

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

➡ String s1 = “Hello”;

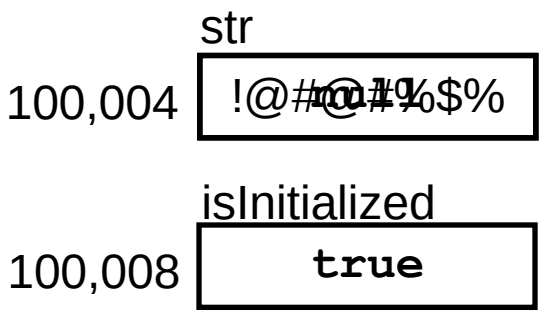
➡ String s2 = “World”;

➡ s1= s2;



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

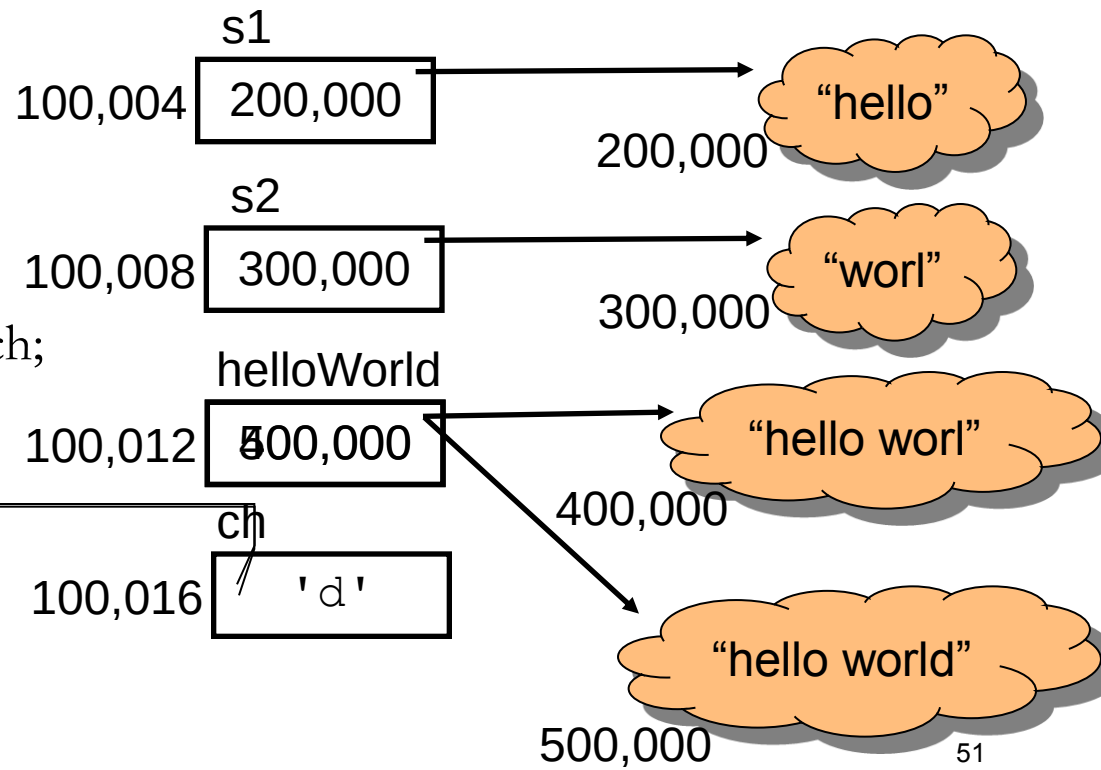


- String str;
- str = null;
- boolean isInitialized = (str == null);

שרשור מחרוזות

- 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;
```



המספר 100 בא מייצג כתוב
(מספר ה unicode) בתא ch? מה כתוב (d

פניה לעצם המוצבע

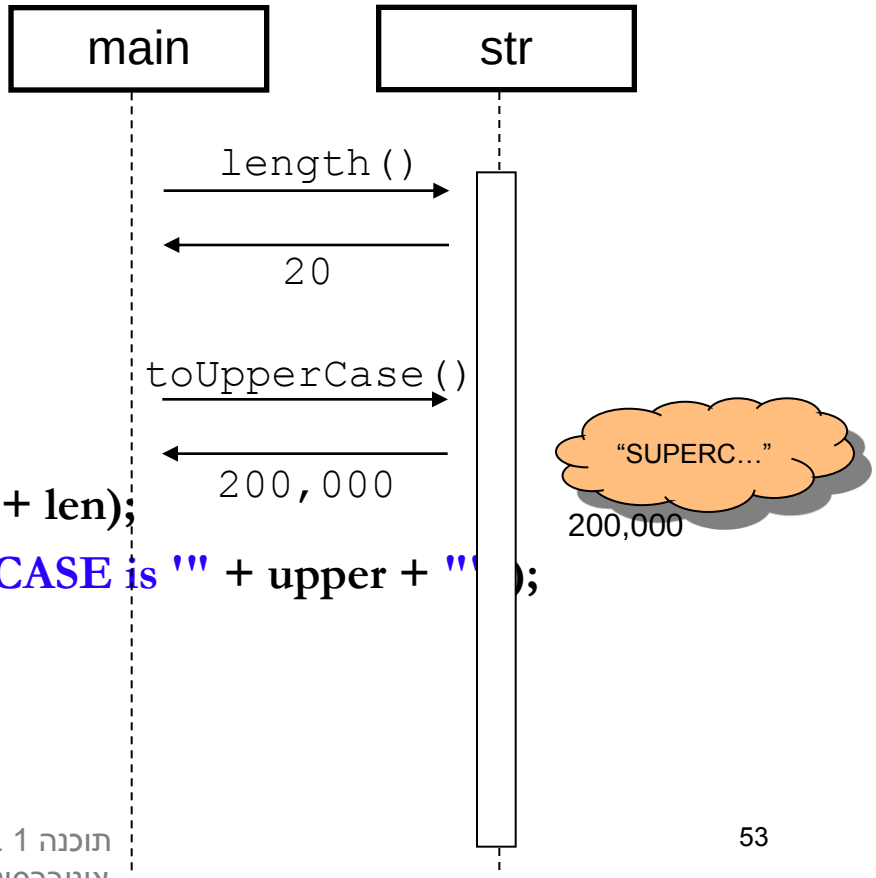
- עד עכשיו כל הפעולות שבצענו היו על ההפנייה. איך ניגשים לעצם המוצבע?
- אופרטור '.' (הנקודה) מאפשר גישה לעצם המוצבע
- מה עושים עם זה?
  - אפשר לבקש **בקשות**
  - אפשר לשאול **שאלות** (ולקבל תשובות)
  - במקרים מסוימים אפשר לגשת **למאפיינים פנימיים** ישירות
- הבקשות השאלות והמאפיינים הפנימיים משתנים מעצם 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

- לעתים יש לנו צורך בסדרת משתנים מאותו טיפוס
- הצורה הפשוטה ביותר לממש זאת היא ע"י מערכים (arrays)
- אנלוגי למבנה Vector בשפת Scheme
- למשל מערך שיכיל את כל המספרים הראשוניים עד למקום מסוים:

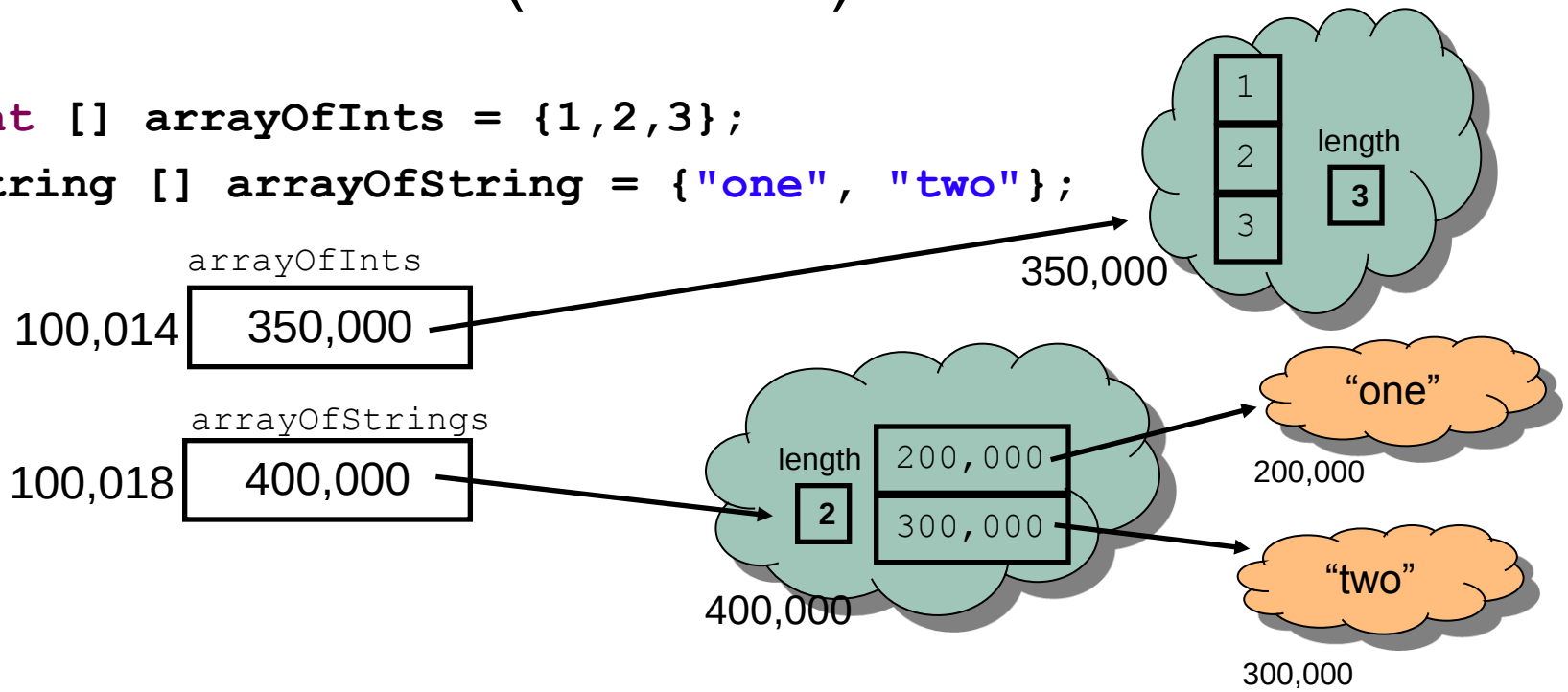
2	3	5	7	11	13	17	19	23	29

- מערכים הם אוסף משתנים מאותו סוג, בין אם פרימיטיבי או התייחסות
- תאים במערך יושבים בדרך כלל ברצף בזיכרון (Java רצה על מכונה וירטואלית!) כך שגישה סידרתית אליהם עשויה להיות יעילה

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"};
```



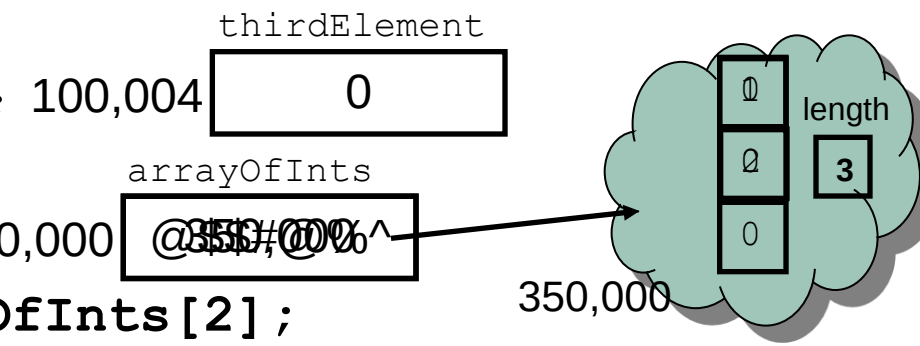
מערכים

- 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];
```

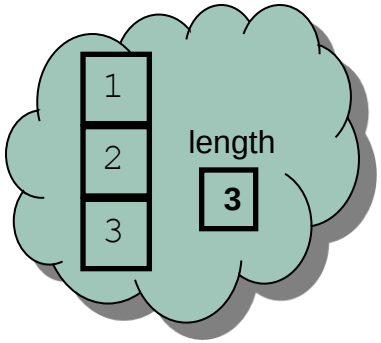


- אברי מערך שהוקצה ע"י `new` מאותחלים אוטומטית לפי טיפוסם:
  - הטיפוסים הפרימיטיביים השלמים מאותחלים ל-0
  - הטיפוסים הפרימיטיביים הממשיים מאותחלים ל- 0.0
  - הטיפוסים הפרימיטיבי `boolean` מאותחלים ל- `false`
  - הטיפוסים הפרימיטיבי `char` מאותחל לתו שערך ה `Unicode` שלו הוא 0
  - טיפוס הפנייה מאותחל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

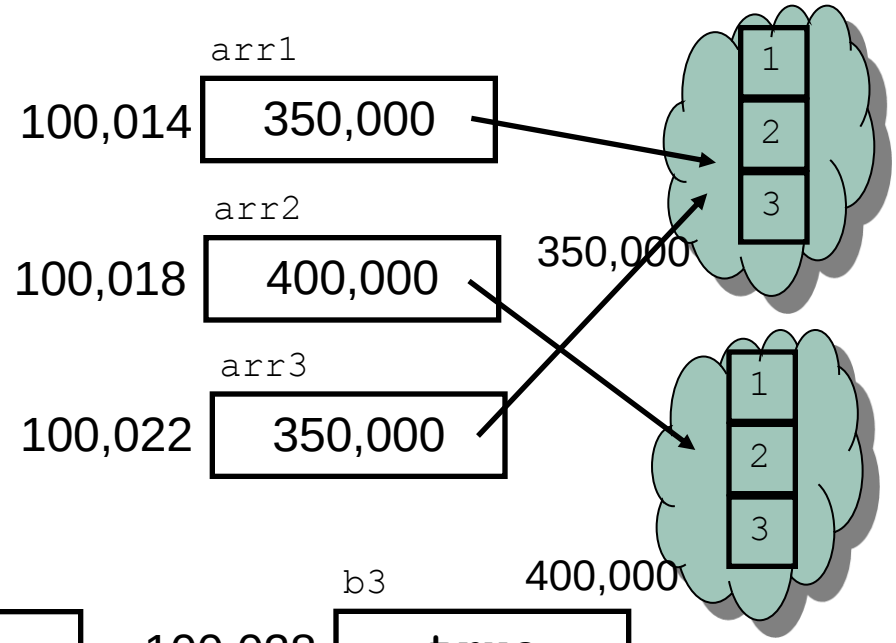


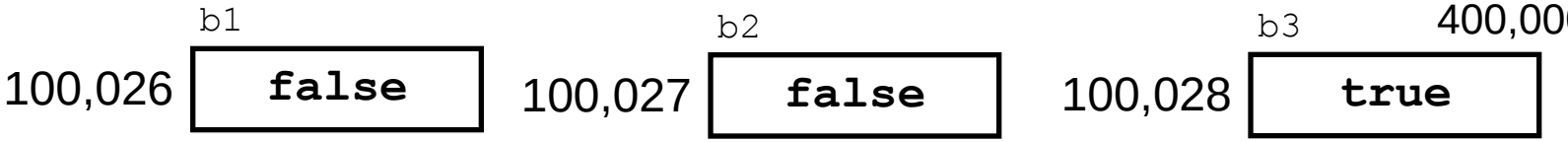
- **תרגיל: איך נדפיס את האיבר האחרון במערך?**

הפניות ואופרטור ההשוואה (==)

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

```
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);
```

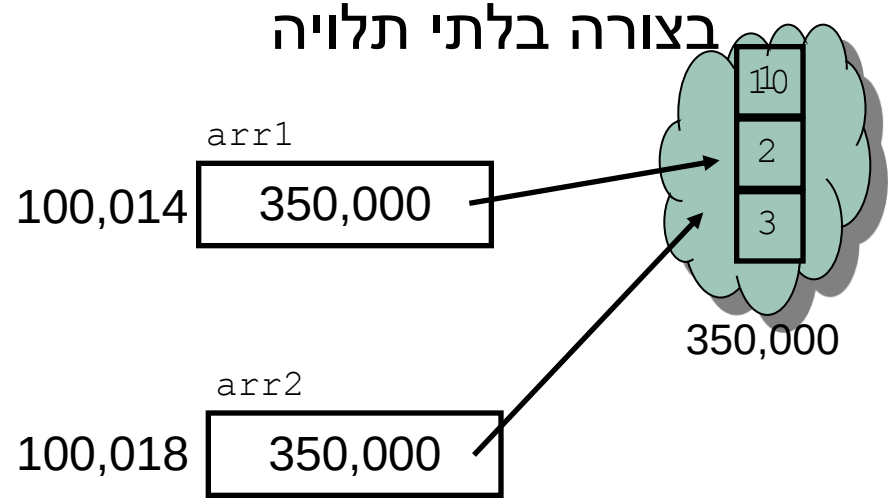




שיתוף (sharing, aliasing)

- 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 (+)

ולקינוח

- חידה (סתם בלי סיבה)
  - אם אתם שוברים מקל באקראי בשני מקומות (ז"א נשארים עם שלושה חלקים ביד), מה הסיכוי שאפשר להרכיב משולש מהשברים?