חלק 1
תכנות פרוצדורלי בג'אווה

דוגמא של תכנית בג׳אוה

מעובד מהספר Java in a Nutshel

```
/**
 * This program computes and prints the
 * factorial of all integers from 1 and 10
 */
public class Factorial {      // ignore now

  public static void main(String[] args) {
    int num;
    for (num = 1; num < 10 ; num++)
      System.out.println(factorial(num));
  }                         // main() ends here
```

```
  private static int factorial(int x) {
    if (x < 0)
      return 0;

    int fact = 1;
    while(x > 1) {
      fact = fact * x;
      x = x - 1;
    }

    return fact;
  }                // factorial() ends here

}                  // The class ends here
```

מה רואים בדוגמא?

- הערות
- הגדרת מחלקה (נתעלם בשלב הזה)
- הגדרת שרות (מתודה method) וארגומנטים (פרוצדורה כעת)
- הצהרה על משתנה (חייבים להגדיר משתנים)
- טיפוס נתונים int למספרים שלמים (לכל משתנה יש טיפוס)
- ביטוי (אריתמטי), קבוע (literal), (ערך של) משתנה, אופרטור
- משפטים (statements): השמה, if (מותנה), for (לולאה), while (לולאה), return
- קריאה לשרות והעברת ארגומנטים
- הערה: זה לא תכנות מונחה עצמים - לא נוצרים עצמים!

תוכניות ג׳אווה

- תוכנית ג׳אווה מחולקת למחלקות; אין שום דבר בתוכנית פרט למחלקות
- המחלקה `Factorial` מוגדרת בקובץ `Factorial.java`
- הקובץ יכול להכיל עוד מחלקות, אבל הן נגישות רק למחלקות אחרות באותו קובץ
- הקומפיילר מתרגם את הקובץ java. לקובץ class.
- נציג עכשיו את עיקרי התחביר של ג׳אווה
- כאשר נציג תחביר של מבנה מסוים, נשתמש בשמות בתוך סוגריים משולשים לציון יחידות תחביריות, לדוגמא <expression>

מבנה לקסיקלי

- תכנית היא סידרה של תווים, הנחלקים ליחידות בסיסיות הנקראות אסימונים (tokens) כגון מספרים, מזהים וכו'.
- אוסף התווים הוא Unicode , שמאפשר ייצוג סימנים משפות שונות (בניגוד ל ASCII, למשל).
- ג'אווה היא case sensitive. לדוגמא המזהה ab שונה מ aB
- ג'אווה מתעלמת מרווחים, סימני טאב, שורה חדשה וכו', פרט לאלה שמופיעים בתוך תווים מצוטטים ומחרוזות ליטרליות. למשל "a string" שונה מ "astring"

הערות

התוכנית מיועדת להיקרא על ידי המחשב (למעשה על ידי הקומפיילר), אבל גם על ידי תוכניתנים

הערות הן טקסט בתוכנית שמיועד רק לקוראים אנושיים

```
/** Returns a specific version */
public String getVersion(int i) {
  Version v = last;
  /* count down the list */
  for (int j = length(); j != i; j--)
    v = v.previous;
  return v.value; // we are done
}
```

סוגי הערות

בג'אווה שלושה סוגי הערות

```
/** doc comment; הערה שעוברת לתיעוד */
/* הערה רגילה, יכולה להתפרס על מספר שורות */
// הערה עד סוף השורה
```

הערות לתיעוד (doc comments) 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. הפורמט לא כולל את תיעוד החוזה (אך ניתן להוסיף).

בשקפים הערות יופיעו בעברית או בגופן מיוחד (*comments*) ללא הסימון המיוחד (/* */ או //)

מילות מפתח בג׳אווה

- המילים המופיעות בשקף הבא הן מילות מפתח (keywords) 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

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

מזהים

- מזהה הוא שם שניתן למרכיב כלשהו של תכנית, כגון מחלקה, שרות, משתנה
- מזהה יכול להיות באורך כלשהו, ולהכיל אותיות ספרות ואת הסימן _ (וכן סימנים נוספים שלא נפרט)
- מזהה אינו יכול להתחיל בספרה
- (שונה מהכללים לגבי מזהים ב scheme , אך דומה לרוב השפות האחרות)
- דוגמאות: my_counter theNumberOfItems x1 n
- מומלץ להשתמש בשמות משמעותיים
- קיימות מוסכמות לגבי סוגי שמות (נראה בהמשך)

קבועים (literals)

- 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: true "abc" 'a' 1.5 3

סימני פיסוק

- סימני פיסוק מופיעים גם הם כאסימונים משני סוגים:
- מפרידים `( ) { } [ ] < > : ; , . @`
- אופרטורים

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

- נראה בהמשך את משמעות האופרטורים, אבל לא את כולם

טיפוסי נתונים (types)

- כל ערך שייך לטיפוס
- כל משתנה בתכנית חייב להיות מוגדר עם טיפוס, ובמהלך התכנית ערכי המשתנה יהיו תמיד מהטיפוס שהוגדר (בשונה מ-scheme)
- כלומר טיפוס היא תכונה סטטית של ערכים, ביטויים וכו׳
- בשפה עם טיפוסים סטטיים ניתן לגלות בזמן קומפילציה שגיאות שקשורות לטיפוסים: למשל, הפעלת פעולת כפל על מחרוזות
- ג׳אווה מגדירה אוסף טיפוסים פרימיטיביים, ומספקת ספריה של מחלקות שמגדירות טיפוסים נוספים (לא פרימיטיביים)
- המתכנת יכול להגדיר טיפוסים נוספים ע״י הגדרת מחלקות

טיפוסי נתונים פרימיטיביים

כזכור, ב-scheme מספרים אינם מוגבלים בגודלם, אין הפרדה בין שלמים לממשיים, ואין טיפוס נתונים נפרד לערכים בוליאניים

ג׳אווה תומכת ב 8 טיפוסי נתונים פרימיטיביים:

- טיפוס בוליאני boolean
- טיפוס של תווים char
- ארבעה טיפוסי מספרים שלמים byte, short, int, long
- שני טיפוסי מספרים בנקודה צפה float, double

הטיפוס הבוליאני

משתנים בוליאניים (`boolean`) יכולים לקבל שני ערכים, `true` ו-`false`. אופרטורים של השוואה, מחזירים ערך בוליאני. לדוגמה:

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

טיפוסים שלמים

- ג'אווה 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 או שליליים (או אפס). הטווח של כל טיפוס נקבע על פי מספר הסיביות בייצוג, למשל 128- עד 127 למשתנים מטיפוס byte
- אין בג'אווה טיפוסים לייצוג מספרים אי שליליים בלבד, כדוגמת unsigned int בשפת C

char לייצוג תוים ושימוש בתו כשלם

- ג'אווה מספקת טיפוס פרימיטיבי לייצוג תווים; תווים הם הסמלים שאנו משתמשים בהם לייצוג טקסט, והם כוללים אותיות (של כל השפות), ספרות, וסימני פיסוק
- בג'אווה תווים מיוצגים על ידי מספרים אי שליליים בייצוג של 16 סיביות, על פי קידוד יוניקוד (Unicode), שמגדיר מספר עבור כל תו. למשל, התו 'A' מקודד על ידי המספר 65, ואילו התו 'א' מקודד על ידי המספר 1488
- קבועים מטיפוס char ניתן לייצג בתוכנית בין גרשיים או באמצעות הערך המספרי,

```
char c = 'A';
c = 1488; //כעת המשתנה מייצג את האות העברית א
```

char (המשך)

- 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 = 'A';
String s = "The letter "+c; // "The letter A"
int i = 65;
String t = "The number "+i; // "The number 65"
```

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

טיפוסים של נקודה צפה

- ג'אווה מספקת שני טיפוסים עבור משתנים שמכילים מספרים ממשיים בייצוג של נקודה צפה:
- `float` 32 סיביות: 1 לסימן, 8 לחזקה (של 2), ו-23 לשבר
- `double` 64 סיביות: 1 לסימן, 11 לחזקה, ו-52 לשבר
- שני הטיפוסים מייצגים מספרים בפורמט סטנדרטי בשם IEEE-754
- ישנם פרטים רבים על הטיפוסים הפרימיטיביים שלא פרטנו; חלקם מופיעים בקובץ הערות שנמצא באתר
- פרטים על ייצוג מספרים וכו' ילמדו בפרויקט תוכנה.

הטיפוס String אינו פרימיטיבי

- לייצוג מחרוזות של תווים משתמשים בטיפוס String
- String הוא טיפוס חשוב, אך אינו פרימיטיבי
- זהו טיפוס שהוא מחלקה, שמוגדרת בספריה סטנדרטית
- מכיוון שמחרוזות הן בשימוש נרחב, ג'אווה מספקת תחביר מיוחד לקבועים (literals) ע"י סימני ציטוט כפולים. לדוגמא

```
"Hello world!"
```

- בהמשך נפרט עוד על מחרוזות

משתנים

- כמו בשפות אחרות, משתנים מכילים ערך שיכול להשתנות במהלך התכנית על ידי השמה
- בג'אווה יש להצהיר על משתנים וההצהרה כוללת את הטיפוס
- ניתן לצרף להצהרה גם אתחול, לדוגמה,

```
int x = 5;          // הצהרה עם אתחול
int y;              // הצהרה בלי אתחול מפורש
String my_string;
```

התייחסויות לעומת ערכים פרימיטיבים

- הטיפוסים בג׳אווה נחלקים לשני סוגים, ובהתאם לכך יש שני סוגי משתנים
- משתנים מהטיפוסים הפרימיטיביים (משתנים פרימיטיביים) אינם מיוצגים על ידי עצם ממחלקה; משתנה כזה מכיל ערך מתחום ערכים מסוים, ואינו יכול שלא להכיל ערך
- משתנים מטיפוסים אחרים יכולים להכיל התייחסות (reference) לעצם ממחלקה מתאימה, או את הערך `null` שמשמעותו שהמשתנה אינו מתייחס כעת לשום עצם

השמה

- השמה (assignment) היא פעולה בסיסית שנותנת ערך למשתנה.
- זהו הבסיס לתכנות אימפרטיבי (בניגוד לתכנות פונקציונלי) שפועל ע"י שינוי ערכי משתנים; תכנות מונחה עצמים בג'אווה נבנה על התשתית של תכנות אימפרטיבי
- אופרטור ההשמה הוא = (דומה ל !set ב-scheme).
- התחביר של השמה הוא:

```
<variable> = <expression>
```

- הביטוי (צד ימין) מחושב, והמשתנה (צד שמאל) מקבל את ערך התוצאה
- השמה למשתנים פרימיטיביים לעומת משתני התייחסות

השמה של ערכים פרימיטביים

השמה מעתיקה ערך פרימיטיבי ממשתנה אחד לאחר

```
int x = 5;
int y = x;
y = 3;
```

הערך של x עדיין 5, משום שהערך שלו רק הועתק למשתנה y ששונה בהמשך

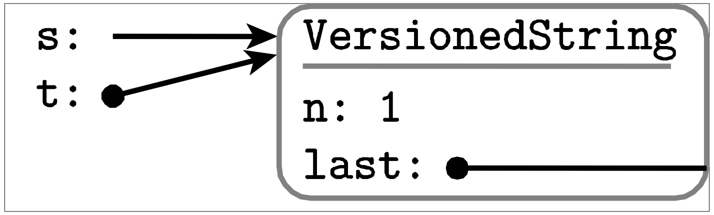
- משתנה פרימיטיבי הוא מיכל לאחסון ערך
- לא כמו נוזל, שמוזגים ממיכל למיכל!

השמה של התייחסויות

השמת התייחסות מעתיקה את ההתייחסות, לא את העצם שמתייחסים אליו:

```
VersionedString s = new VersionedString();
s.add("V 1");
VersionedString t = s;
```

אחרי ההשמה, שני המשתנים מתייחסים לאותו עצם בדיוק:



שינוי מצב העצם דרך t משנה את העצם ש-s מתייחס אליו:

```
t.add("V 2");
s.getLastVersion(); // returns "V 2"
```

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)

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	`== !=`
וגם (לערכים בוליאניים או שלמים כווקטורי סיביות)	`&`
או אקסקלוסיבי (כנ"ל)	`^`
או (כנ"ל)	`|`
וגם בוליאני "קצר" (מתעלם מהאופרנד השני אם הראשון כבר קובע את התוצאה)	`&&`
או בוליאני "קצר"	`||`

אופרטורים אונריים

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

- האופרטורים מסודרים לפי סדר קדימות, אבל פרט לשורה הראשונה יש להם אותה קדימות.
- האופרטורים האונריים קודמים לבינריים
- אופרטורים בינריים אסוציאטיביים לשמאל, אונריים לימין (פרט ל `x++  x--`)

אופרטור התנאי

אופרטור התנאי דומה לביטוי if ב scheme . התחביר:

```
<boolean-expression> ? <t-val> : <f-val>
```

- ראשית, הביטוי <boolean-expression> (ביטוי התנאי, שחייב להיות בוליאני) מחושב
- אם ערכו true, מחושב ומוחזר ערך הביטוי <t-val>
- אם ערכו false, מחושב ומוחזר ערך הביטוי <f-val>
- דוגמא:

```
System.out.print(n==1 ? "child" :"children");
```

- הקדימות שלו היא אחרי האופרטורים הבינריים

השמה עם פעולה

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

x += y; *is equivalent to*

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. השמה מרובה x = y = <exp>

אופרטורים: קדימות ואסוציאטיביות

השפה מגדירה חוקי קדימות ואסוציאטיביות לאופרטורים

החוקים מתנהגים בדרך כלל באופן צפוי, למשל

```
x + y * z   ≡   x + ( y * z )    קדימות
x + y + z   ≡   (x + y ) + z     אסוציאטיביות
x = y = z   ≡   x = ( y = z )    אסוציאטיביות
```

אבל במקרים שאינם מובנים מאליהם, כדאי להשתמש בסוגריים (בפרט אם צריך להיעזר בטבלת הקדימויות/אסוציאטיביות; אם אתם זקוקים לטבלה, גם מי שקורא את הקוד יהיה זקוק לה):

```
x + y >> z  ≡   ???
```

אולי צריך סוגריים ואולי לא צריך, אבל בכל מקרה כדאי

משפטים

- משפט הוא פקודה לביצוע (בעיקר לצורך תוצאי הלוואי, שינוי ערכים של משתנים). כל משפט מסתיים בנקודה-פסיק ;
- המשפט הבסיסי הוא השמה
- משפטים מתבצעים בזה אחר זה, אלא אם מדובר במשפטי בקרה, שנועדו לקבוע סדר, בחירה או חזרה
- משפטים מותנים (משפטי בחירה): משפט `if/else`, משפט `switch`
- משפטי חזרה (לולאות): משפט `while`, משפט `do`, משפט `for`
- כדי לקבץ ביחד מספר משפטים, בעיקר לצרכי בקרה, משתמשים במשפט מורכב שנקרא גוש (block)

גושי פסוקים

גושי משפטים דומים ל begin או let ב scheme.

לעיתים קרובות צריך להשתמש במבנה בקרה כדי לשלוט על ביצוע של גוש פסוקים שלם ולא פסוק בודד

גוש כזה מסומן על ידי הקפה בסוגריים מסולסלים

```
i = n;
while ( i > 0 ) {
  f = f*i;
  i--;
}
```

הצהרה על משתנים בתוך גוש

למשתנה שמוצהר בתוך גוש ניתן לגשת רק בתוך הגוש

הצהרה בגוש כוללת הצהרה בפסוק האתחול של לולאת for

```
int s = 0;
for ( int i = 1; i < n; i++ ) {
  int j = i*i;
  s = s + j;
}
System.out.println("j="+j);  error, j undeclared
System.out.println("i="+i);  error, i undeclared
System.out.println("s="+s);  OK
```

משפט if/else

- דומה למשפט if ב-scheme
- דוגמאות:

```
if (username == null)
   username = "John Doe";

if ((x > 0) && (x <= 10)) {
   y = x * x;
   z = x + 3;
}
```

משפט if/else (המשך)

```
if (x == y)
    z = x + 1;
else
    z = x + y;
```

תחביר:

- התנאי בתוך סוגריים
- פסוק ה else הוא אופציונלי
- שימוש בגוש כאשר יש לבצע יותר מפקודה אחת במקרה שהתנאי מתקיים ו/או כאשר אינו מתקיים
- מה ההבדל בין זה לאופרטור התנאי?

קינון משפטי if

```
if (x == y)
    if (y == 0)
        System.out.println(" x == y == 0");
else
        System.out.println(" x == y, y!= 0");
```

- ה else משויך ל if הקרוב לו; בדוגמא זאת ההיסט (אינדנטציה) מטעה!!
- יש להשתמש ב-{} כדי לשנות את המשמעות, וגם לצרכי בהירות

פיצול למספר מקרים בקינון if

```
if (exp1) {
      // code for case when exp1 holds
}
else if (exp2) {
// when exp1 doesn't hold and exp2 does
}
// more...
else {
// when exp1, exp2, ... do not hold
}
```

משפט switch - דוגמא

```
switch (n) {
  case 1 :
    // code for the case that n == 1
    break;
  case 2 :
    // code for the case that n == 2
    break;
  default:
    // code for the other cases
}
```

משפט switch (המשך)

- חלופה אפשרית ל-if מקונן, אבל לא בכל מקרה
- משמש לפיצול בין מספר מקרים לפי ערך של ביטוי. מבנה:

```
switch (<expr>) {
   <case-expr> : <statement> // several
    ...
   default : <statement> // optional, last
}
```

- הביטוי <expr> חייב להיות מאחד הטיפוסים int, short, byte, char או טיפוס מנייה (נלמד בהמשך)
- הביטויים <case-expr> ("תוויות") חייבים להיות (ביטויים) קבועים מהטיפוס המתאים, ושונים זה מזה

משפט switch (המשך)

- הביטוי <expr> מחושב, והביצוע ממשיך אחרי התווית עם הערך המתאים
- אם ערך הביטוי אינו מופיע כאחת התוויות, הביצוע ממשיך אחרי תווית המחדל (או מדלג על כל המשפט אם אין תווית ערך מחדל)
- כדי להשתמש ב-`switch` לביצוע פיצול, יש להשתמש במשפטי ה-`break`, אחרת הביצוע "יזרום" למקרה הבא.
- ניתן לכתוב מספר תוויות זו אחר זו עבור אותו משפט:

```
switch (response)
   case 'Y':
   case 'y': answer = true; break; ...
```

משפט switch (דוגמה)

```
System.out.print("You won ");
switch ( n ) {
  case 1:  // חייב להיות קבוע מספרי שלם
    System.out.println("a medal");
    break; // בלעדי הפסוק נבצע גם את מקרה 2
  case 2:
    System.out.println("a pair of medals");
    break;
  default:
    System.out.println("many medals");
}
```

לולאות

- בקורס המבוא למדנו על תהליכים איטרטיביים ורקורסיביים; שניהם נכתבו בתחביר של רקורסיה (האינטרפרטר ממיר רקורסית זנב לאיטרציה)
- בג׳אווה, כמו ברוב השפות, איטרציה כותבים במפורש בעזרת משפטים מיוחדים שנקראים לולאות
- ג׳אווה מאפשרת גם רקורסיה, בצורה הרגילה (כאשר שרות קורא לעצמו, ישירות או בעקיפין)
- ג׳אווה תומכת בשלושה סוגים של לולאות: משפט while, משפט do , ומשפט for

משפט while

```
while ( <expression> )
    <statement>
```

הביטוי <expression> (תנאי הלולאה) צריך להיות בוליאני.

ביצוע משפט ה while נעשה כך:

1. הביטוי <expression> מחושב.
2. אם ערכו false מדלגים על <statement> (גוף הלולאה)
3. אם ערכו true מבצעים את גוף הלולאה, וחוזרים למס' 1

```
while ( v != null )
  v = v.previous;
```

משפט do

```
do
  <statement>
while ( <expression> ) ;
```

- כאן התנאי מחושב לאחר ביצוע גוף הלולאה
- לכן הלולאה מתבצעת לפחות פעם אחת.
- לפעמים מאפשר לחסוך כתיבת שורה לפני הלולאה

```
do
  v = v.previous;
while ( v != null )
```

משפט for

```
for (<initialize> ; <test> ; <increment> )
    <statement>
```

- לולאת for שקולה ללולאת ה while הבאה

```
<initialize> ;
while <test> {
    <statement> ;
    <increment>
}
```

- ניתן לכלול ב <initialize> הגדרה של משתנה שתחום הקיום שלו הוא הלולאה.

לולאות לדוגמא

לולאת ה for :

```
for (int count = 0; count < 10; count++;)
    System.out.println(count);
```

שקולה ללולאה הבאה.

```
int count = 0;
while (count < 10) {
    System.out.println(count);
    count++;
}
```

במקרה זה לולאת ה for קריאה יותר.

לולאת for (המשך)

- החלקים <initialize> ו <increment> יכולים להכיל יותר מביטוי אחד, מופרדים בפסיקים. לדוגמא:

```
for (int i = 0, j = 10; i < 10 ; i++, j-- )
    sum += i * j;
```

- כאן יש שתי הצהרות של שלמים (תחביר כללי לסדרת הצהרות מופרדות בפסיק, של משתנים מאותו טיפוס).
- ה <increment> , למרות שמו, יכול לטפל לא רק בהגדלת מספרים, אלא גם (למשל) להתקדם בסריקה של מבנה נתונים (דוגמאות בהמשך).
- נראה בהמשך צורה נוספת של לולאה לצורך מעבר על מבנים מורכבים כגון מערכים או אוספי עצמים.

משפט break

- ביצוע משפט זה גורם ליציאה מיידית מהמבנה המכיל אותו (משפט while, do, for, switch).

```
break;
```

- ראינו דוגמא במשפט switch, שם זה שימושי מאד
- דוגמא נוספת:

```
for ( int i = 1; i < 1000; i++ ) {
  int j = i*i;
  if (j > 1000) break;
  s = s + j;
}
```

משפט continue

- ביצוע משפט זה גורם ליציאה מיידית מהאיטרציה הנוכחית של לולאה, ולהתחיל מייד את האיטרציה הבאה.
- יכול להופיע רק בתוך לולאה (משפט while, do, for).

```
for ( int i = 1; i < 100; i++ ) {
  if ( i % 17 == 0 )
      continue;
  s = s + i;  סוכם את השלמים שלא מתחלקים ב-17
}
```

משפט break ו continue

- קיימת צורה נוספת של `break` שכוללת תווית, ומאפשרת לצאת ממבנה כלשהו, לאו דווקא המבנה העוטף הסמוך ביותר, ולאו דווקא לולאה או switch
- קיימת גם גירסא של `continue` עם תווית
- פרטים בקובץ ההערות שנמצא באתר הקורס

משפטים וביטויים

- ביטויים (expressions) הם מבנים שחישובם נותן ערך
- משפטים (statements) הם פקודות שביצוען נועד בעיקר לתוצא לוואי (שינוי ערך, פעולת קלט/פלט).
- אבל ההפרדה אינה מלאה
- לחלק מסוגי הביטויים יש תוצא לוואי (side effect)
- ביטויים כאלה יכולים להופיע כמשפט. במקרה זה רק תוצא הלוואי חשוב - הערך שהביטוי מחזיר "נזרק לפח":

```
i++;
```

- ניתן להשוות ל `m = i++;`
- ולעומת זאת, `m = ++i;`
- שתי הדוגמאות האחרונות מבלבלות; עדיף להימנע מהן

שרות

- בדומה לשיגרה (פרוצדורה) בשפות תכנות אחרות, שרות הוא סדרת משפטים שניתן להפעילה ממקום אחר בקוד של ג'אווה ע"י קריאה לשרות, והעברת אפס או יותר ארגומנטים
- כרגע נדון רק בשירותים שהם פונקציות או פרוצדורות (יש בג'אווה עוד סוגי שירותים); שירותים כאלה מוכרזים על ידי מילת המפתח static, כמו למשל

```
public static int fibonaci(int n) {
  if (n==0 || n==1) return 1;
  return fibonacci(n-1)+fibonacci(n-2);
}
```

- נתעלם כרגע מההכרזה public

הגדרת שרות

- התחביר של הגדרת שרות הוא:

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

- ה <modifiers> הם 0 או יותר מילות מפתח מופרדות ברווחים (נראה בהמשך)
- <type> מציין את טיפוס הערך שהשרות מחזירה. void מציין שהשרות אינו מחזיר ערך.
- <paramlist> רשימת הפרמטרים הפורמליים, מופרדים בפסיק, וכל 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` והשרות יסתיים כאשר הביצוע יגיע לסופו

גוף השרות

- גוף השרות מכיל הצהרות על משתנים זמניים (variable declarations) ופסוקים ברי ביצוע (כולל return)
- הצהרות יכולות להכיל פסוק איתחול בר ביצוע

```
public String getVersion(int i) {
  Version v = last;
  for (int j = length(); j != i; j--)
    ...
```

- 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

אתחול משתנים זמניים

האם השרות הבא עוברת קומפילציה?

```
public void test(char c) {
  int i;              // אין איתחול
  int x = 3453;
  int d = x/c;
  int r = x%c;
  if (d*c + r == x) i = 1; // התנאי נכון תמיד!!
  System.out.println("i = "+i);
}  // למרות זאת, שגיאת קומפילציה
```

קריאה לשרות

- קריאה לשרות מסוג static שאינו מחזיר ערך (טיפוס הערך המוחזר הוא void) תופיע בתור משפט (פקודה), כמו

```
add("V 2");
```

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

```
num = fact(m+3) + 5;
System.out.println(vs.getVersion(1));
```

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

העברת ארגומנטים

- כאשר מתבצעת קריאה לשרות, ערכי הארגומנטים נקשרים לפרמטרים הפורמליים של השרות לפי הסדר, ומתבצעת השמה לפני ביצוע גוף השרות.
- בהעברת ערך פרימיטיבי הערך מועתק לפרמטר הפורמלי :

```
void f(int y) { y = 3; }
int x = 5;
f(x);  //  x still contain 5; equivalent to { y=x; y=3; }
```

- העברת התייחסות כארגומנט מעתיקה את ההתייחסות, לא את העצם שמתייחסים אליו
- צורה זאת של העברת פרמטרים נקראת call by value

מערכים

```
int[] primes;                // הצהרה
primes = new int[37];   // הקצאה (יצירה) והשמה
```

ביצירה אברי המערך מאותחלים לערכי המחדל של טיפוס האיבר

```
primes[0] = 1;               // האינדקס הראשון הוא 0
...
for (int i=0;
     i<primes.length;   // המערך "יודע" את אורכו
     i++)
  System.out.println(primes[i]
                          +" is a prime");
```

- דומה לוקטורים ב-scheme

התייחסויות למערכים

```
int[] primes;                          הצהרה על התייחסות למערך
primes = new int[37];
computePrimes(primes);                 השרות ימלא את המערך
System.out.println(primes[4]);         ידפיס 7
int[] old_primes = primes;             התייחסות חדשה למערך
primes = new int[100];                 השמה מחדש של ההתייחסות
System.out.println(primes[4]);         ידפיס 0
```

האם ניתן לכתוב שרות שימלא איבר במערך (למשל החמישי)?

```
computeAPrime(primes[4],4);            לא!! זהו משתנה
```

פרימיטיבי, לכן מועבר מספר, לא התייחסות למערך שניתן לשנות

אין בג'אווה מערכים רב מימדיים

אבל יש מערכים של התייחסויות למערכים,

```
double[][] matrix = new double[10][];
for (i=0; i<matrix.length; i++)
  matrix[i] = new double[10];
```

זהה ל:

```
double[][] matrix = new double[10][10];
```

מטריצה משולשית תחתונה

```
double[][] tri = new double[10][];
for (i=0; i<matrix.length; i++)
  tri[i] = new double[i+1];
```

`tri[7][3] ≡ ( tri[7] ) [3]` אסוציאטיביות לשמאל

הגדרת מערכים ומערכים אנונימיים

```
int[] primes = { 1, 2, 3, 5, 7, 11, 13 };
```

ההגדרה יכולה להשתמש בערכים מחושבים, לא רק קבועים,

```
int[] primes = { getPrime(1),
                 getPrime(2),
                 ...
                 getPrime(7) };
```

לשרות אפשר להעביר התייחסות למערך אנונימי,

```
printPrimes( new int[] { 1, 2, 3, 5, 7 } );
```

תכנית ג׳אווה

```java
public class PrintPrimes {
   public static void main(String[] args) {
      int primes = new int[10];
      ...
      for (int i; i<10; i++)
         System.out.println(
            "The "+i+"th prime is "+primes[i]);
   }
}
```

קומפילציה והרצה

```
c:\temp> javac PrintPrimes.java
c:\temp> java -cp . PrintPrimes
2 3 5 7 11 13 17 19 23 29
```

- הפקודה הראשונה מהדירה (מתרגמת, compiles) את התוכנית ששמורה בקובץ PrintPrimes.java לשפת מכונה מיוחדת לג׳אווה; התוצאה נשמרת בקובץ PrintPrimes.java
- הפקודה השניה מריצה את השירות main (תמיד) של המחלקה PrintPrimes
- בסביבת הפיתוח שלנו (Eclipse) קומפילציה מתבצעת אוטומטית כאשר שומרים את הקובץ (save), ואפשר להריץ את התוכנית בעזרת תפריט ההקשר (עכבר ימני)

עוד על תוכניות ג'אווה

- מחלקות מאוגדות בדרך כלל בחבילות (packages)
- המחלקה VersionedString בחבילה il.ac.tau.cs.oopj מוגדרת בקובץ il/ac/tau/cs/oopj/VersionedString.java
- למחלקה מחבילה אחרת, ניתן להתייחס בשמה המלא, למשל java.util.Stack (מחלקה יותר כללית מ `IntStack`)
- לחליפין, ניתן לייבא מחלקה, או את כל המחלקות מחבילה, ולהשתמש בשם הקצר (זהירות! שמות עלולים להתנגש):

```
import java.util.Stack;        // יופיע לפני הגדרת המחלקה
import java.util.*;            // כל המחלקות בחבילה
...  Stack stk
```