תוכנה 1 בשפת Java

שיעור מספר 2: "תזרום אח שלו"

בית הספר למדעי המחשב

אוניברסיטת תל אביב

מה בתכנית?

- מבני בקרה ב Java
  - משפטי תנאי
    - if
    - switch
  - לולאות
    - while/do-while
    - for
  - זימון והגדרת שרותי מחלקה (methods)
  - העמסת שרותים

התקדמות לינארית

- בשיעור הקודם דנו בביטויים בשפה – וראינו תוכניות פשוטות שמבצעות פעולות (בעיקר השמה) על ביטויים ומשתנים
- התוכנית התקדמה באופן קווי (לינארי) – החל בפונקציה **`main`** התוכנית בצעה שורה אחר שורה את התוכנית, עד שהגיעה לסוף **`main`** והסתיימה
- ציינו בעזרת החץ ⇨ את השורה הנוכחית שבה אנו נמצאים. בסוף כל שורה ירד החץ שורה למטה
- איך מיוצג החץ במודל הזכרון של התוכנית?

מצביע התוכנית (program counter)

- לפני תחילת ריצת התוכנית מועתק קוד התוכנית לאיזור מסוים בזכרון המחשב (איזור בשם CODE)
- משתנה מיוחד בשם *מצביע התוכנית* (program counter) מציין באיזו שורה אנחנו נמצאים בביצוע הקוד
- לאחר ביצוע כל שורה בתוכנית מעודכן ערך המצביע לשורה הבאה שאותה יש לבצע
- נציג היום כמה משפטים ב Java אשר משפיעים על *זרימת התוכנית*. כלומר גורמים לתוכנית להתבצע בצורה שאינה לינארית

משפט if

- מזכיר ביטוי if ב – scheme
- דוגמא:

```
...
if (grade > 60)
   System.out.println("You passed the test!");
System.out.println("Your grade is: " + grade);
```

- הביטוי בסוגריים חייב להיות ערך בולאני
  - אם ערכו הוא **true** יתבצע המשפט או הבלוק המופיע מיד אחרי הסוגריים (פסוקית-אז, then-clause), ואחר כך תמשיך התוכנית בצורה סדרתית
  - אחרת, התוכנית תדלג על משפט (או בלוק) זה

- מה יודפס עבור grade שהוא 40?
- מה יודפס עבור grade שהוא 100?

משפט if (עם בלוק משפטים)

- **בלוק** הוא אוסף משפטים עטוף בסוגריים מסולסלים
- דוגמא:

```
...
if (grade > 60) {
   System.out.println("You passed the test!");
   System.out.println("Hip Hip Hooray !");
}
System.out.println("Your grade is: " + grade);
```

- הלוגיקה של זרימת התוכנית מתקיימת עבור הבלוק כפי שהתקיימה עבור המשפט הבודד בשקף הקודם

- מה יודפס עבור grade שהוא 40?
- מה יודפס עבור grade שהוא 100?

משפט if/else

- אם נרצה שכאשר התנאי מתקיים יתבצעו משפטים מסוימים וכאשר הוא אינו מתקיים יתבצעו משפטים אחרים נשתמש במשפט else

```
...
if (grade>60) {
   System.out.println("You passed the test!");
   System.out.println("Hip Hip Hooray !");
} else {
   System.out.println("You failed");
   System.out.println("It is so sad...");
}
System.out.println("Your grade is: " + grade);
```

- אם התנאי אינו מתקיים התוכנית מדלגת על ביצוע פסוקית-אז וקופצת לשורת ה else
- משפט else יכול להכיל משפט אחד, בלוק או לא להופיע כלל
- משפט else הוא בעצם פסוקית (פסוקית-אחרת, else-clause) - הוא יכול להופיע רק אחרי פסוקית-אז

- מה יודפס עבור grade שהוא 40?
- מה יודפס עבור grade שהוא 100?

משפט if/else מקוננים (nested if)

- הביטוי הבוליאני ב- if יכול להיות מורכב:

```
if (grade>=55 && grade<60)
   // recheck the exam...
```

- פסוקית-אז ופסוקית-אחרת יכולים להיות בעצמם משפטי if:

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

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

אופרטור התנאי

- אופרטור התנאי (להבדיל מ**משפט תנאי**) 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 <t-val> מחושב ומוחזר כערך הביטוי
  - אם ערכו false <f-val> מחושב ומוחזר כערך הביטוי

- הקדימות שלו היא אחרי האופרטורים הבינריים
- אופרטור התנאי הוא ביטוי – ניתן לחשב את ערכו, לעומת משפט תנאי שהוא משפט בקרה המשפיע על זרימת התוכנית

אופרטור התנאי

- שימוש באופרטור התנאי:

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

- שימוש במשפט תנאי:

```
if (n==1)
   System.out.print("child");
else
   System.out.print("children");
```

- מה ההבדל?

ריבוי תנאים (else-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
}
```

- למבנה `else if` אין סמנטיקה מיוחדת, נשתמש בו במקרה שמתוך אוסף מקרים אמור להתקיים מקרה אחד לכל היותר
- מה המשמעות של הקוד למעלה ללא ה-`else`?

ריבוי תנאים (switch)

```
System.out.print("Your grade is: ");

switch(grade){
    case 100:
      System.out.println("A+");
    case 90:
      System.out.println("A");
    case 80:
      System.out.println("B");
    case 70:
      System.out.println("C");
    case 60:
      System.out.println("D");
}
...
```

- קיים תחביר מיוחד לריבוי תנאים:
- ארגומנט ה **`switch`** הוא שלם שאינו long או טיפוס מניה (יוסבר בהמשך הקורס)
- מתבצעת השוואה בינו ובין כל אחד מערכי ה **`case`** ומתבצעת קפיצה לשורה המתאימה אם קיימת
- לאחר הקפיצה מתחיל ביצוע סדרתי של המשך התוכנית, תוך התעלמות משורות ה **`case`**
- מה יודפס עבור grade שהוא 60?
- מה יודפס עבור grade שהוא 70?

ריבוי תנאים (break)

```
System.out.print("Your grade is: ");

switch(grade){
    case 100:
       System.out.println("A+");
       break;
    case 90:
       System.out.println("A");
       break;
    case 80:
       System.out.println("B");
       break;
    case 70:
       System.out.println("C");
       break;
    case 60:
       System.out.println("D");
       break;
}
...
```

- ניתן לסיים משפט **switch** לאחר ההתאמה הראשונה, ע"י המבנה **break**
- מה יודפס עבור grade שהוא 70?
- מה יודפס עבור grade שהוא 50?
- שימוש במבנה **default** (ברירת מחדל) נותן מענה לכל הערכים שלא הופיעו ב **case** משלהם
- מקובל למקם את ה **default** כאפשרות האחרונה

ריבוי תנאים (default)

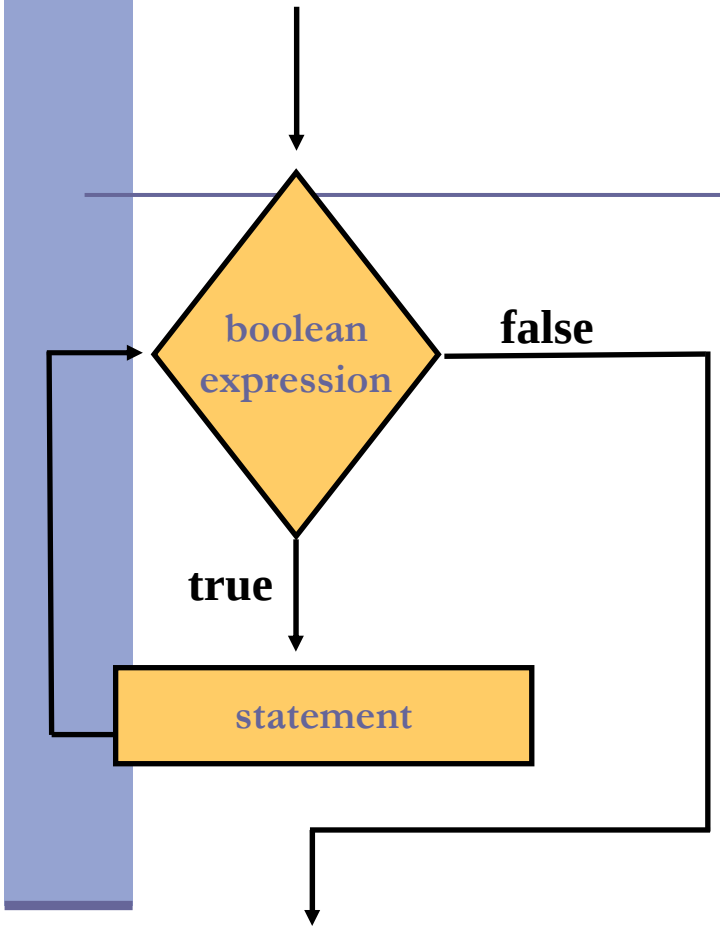
```
System.out.print("Your grade is: ");

switch(grade){
     case 100:
        System.out.println("A+");
        break;
     case 90:
        System.out.println("A");
        break;
     case 80:
        System.out.println("B");
        break;
     case 70:
        System.out.println("C");
        break;
     case 60:
        System.out.println("D");
        break;
     default:
        System.out.println("F");
}
...
```

- מה יודפס עבור grade שהוא 50?
- בתכנות מונחה עצמים נשתדל להימנע משימוש בswitch
- בהמשך הקורס נראה מנגנונים אחרים של השפה, שנחשבים טובים יותר להתמודדות עם ריבוי תנאים

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 ( <boolean_expression> )
   <statement>
```

- ביצוע משפט ה while נעשה כך:

1. הביטוי <boolean_expression> מחושב:
   - אם ערכו **false** מדלגים על <statement> (גוף הלולאה - משפט או בלוק משפטים)
   - אם ערכו **true** מבצעים את גוף הלולאה וחוזרים ל- (1)

- לדוגמא: נשתמש בלולאת **while** כדי להדפיס 1000 פעמים את המחרוזת "אין מדברים בזמן השיעור"

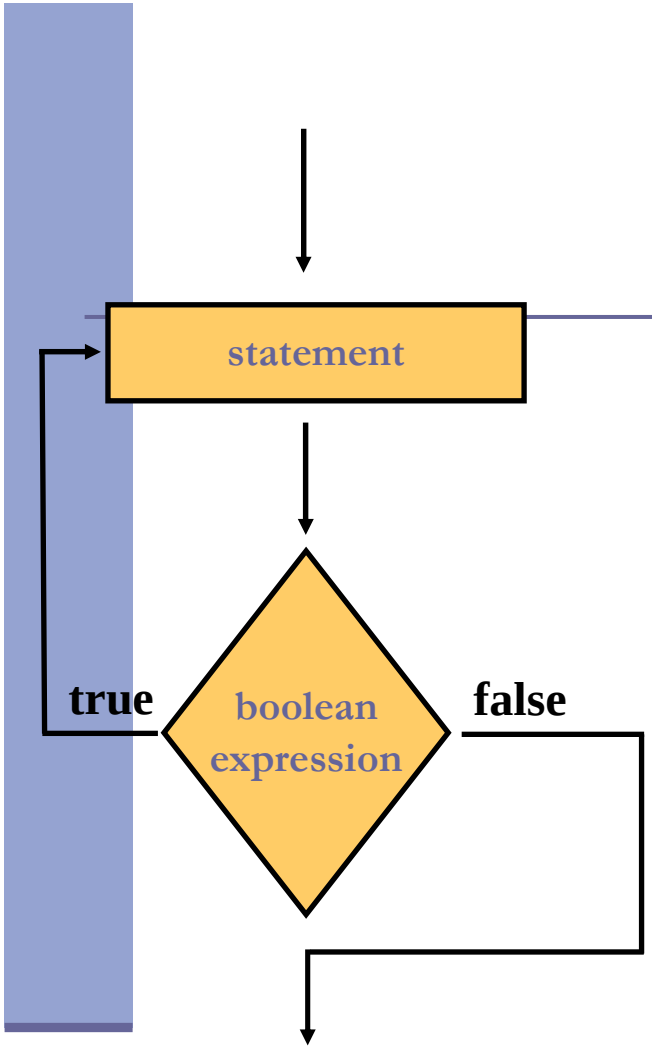
"אין מדברים בזמן השיעור"

```
int counter = 0;
while (counter<1000) {
   System.out.println("No talking during class");
   counter++;
}
```

- אף על פי שהדוגמא פשוטה, נמצאים בה כל מרכיבי הלולאה:
  - הגדרת משתנה עזר ואתחולו
  - בדיקת תנאי עצירה (שנמשיך?)
  - ביצוע איטרציה נוספת
  - קידום משתנה העזר

- מוסכמות:
  - משתני עזר מאותחלים ל- 0
  - בדיקת הסיום היא בעזרת האופרטור < על מספר האיטרציות המבוקש

משפט do




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

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

- נתרגם את לולאת ה- **do-while** מהשקף הקודם ללולאת **while**

משפט do

```
int counter = 0;
do {
   System.out.println("No talking during class");
   counter++;
} while (counter<1000);
```

- הבחירה בין השימוש במשפט **do** לשימוש במשפט **while** (במקרים שבהם ידוע בוודאות שיהיה לפחות מחזור אחד של הלולאה) היא עניין של טעם אישי

for משפט

- במשפט ה **while** ראינו את 4 יסודות הלולאה
- אולם תחביר הלולאה כפי שמופיע ב **while** אינו תומך ישירות בהגדרת משתנה עזר, באתחולו ובקידומו
- המתכנתת צריכה להוסיף קוד זה לפני הלולאה או בתוכה
- תחביר משפט **for** כולל את 4 יסודות הלולאה:

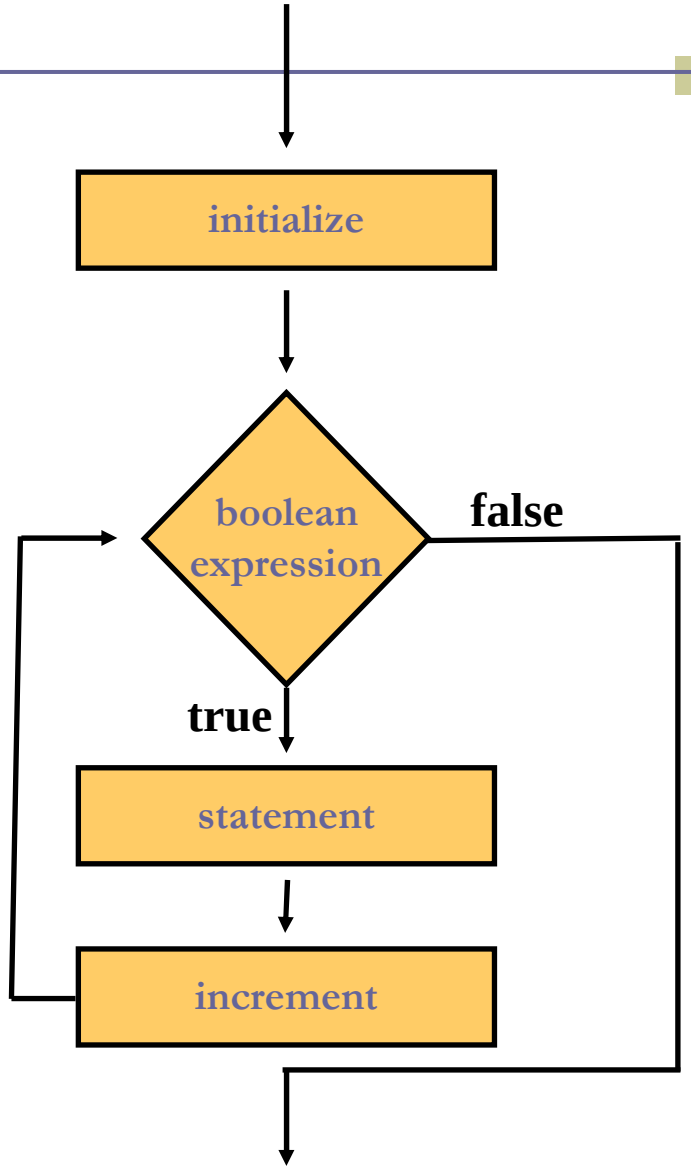
```
for (<initialize> ; <boolean_expression> ; <increment> )
        <statement>
```

for משפט

```
for (int counter = 0; counter<1000; counter++) {
  System.out.println("No talking during class");
}
```

- הגדרת משתנה עזר ואתחולו – משתנה זה מוגדר אך ורק בתחום לולאת ה for ואינו נגיש לאחריו
- בדיקת תנאי עצירה (שנמשיך?)
- קידום משתנה העזר
- גוף הלולאה - ביצוע איטרציה נוספת

for (<initialize> ; < boolean_expression> ; <increment>)
<statement>



משפט for

- ללולאת ה **for**:

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

כמעט שקולה ללולאת ה- **while** :

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

for משפט

- החלקים <increment> ו <initialize> יכולים להכיל יותר ממשפט אחד, מופרדים בפסיקים. לדוגמא:
  - שתי הצהרות של שלמים (תחביר כללי לסדרת הצהרות מופרדות בפסיק, של משתנים מאותו טיפוס):

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

  - הדפסה במשפט <initialize> :

```
int counter;
for(counter = 0 , System.out.println("Starting a loop"); counter<1000; counter++) {
    System.out.println("No talking during class");
}
```

- למרות שמו, יכול <increment> לטפל לא רק בהגדלת מספרים, אלא גם (למשל) להתקדם בסריקה של מבנה נתונים (דוגמאות בהמשך)

משפט break

- ביצוע משפט break גורם ליציאה מיידית מהמבנה המכיל אותו (משפט while, do, for, switch)
- ראינו דוגמא במשפט switch , שם זה שימושי מאד
- דוגמא נוספת – מה עושה הקוד הבא:

```
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 הלולאה, ולהתחיל מייד את האיטרציה הבאה
- יכול להופיע רק בתוך לולאה (משפט while, do, for)

- דוגמא - מה עושה הקוד הבא?

```
for ( int i = 1; i < 100; i++ ) {
   if ( i % 17 == 0 )
      continue;
   s = s + i;
}
```

ההבדל בין continue ל- break

- נציג שתי גרסאות למשחק "2-בום"

```
for ( int i = 1; i < 10; i++ ) {
   if ( i % 2 == 0 )
       break;
   System.out.println(i);
}
System.out.println("The End");
```

- "נריץ" את גרסת ה- break

ההבדל בין continue ל- break

- גרסת continue למשחק "2-בום"

```
for ( int i = 1; i < 10; i++ ) {
   if ( i % 2 == 0 )
       continue;
   System.out.println(i);
}
System.out.println("The End");
```

- וכו'...

לולאות מקוננות

- גוף הלולאה יכול להיות לולאה בעצמו
- נדפיס את לוח הכפל:

```
for ( int i = 1; i <= 10; i++ ) {
   for ( int j = 1; j <= 10; j++ )
      System.out.print(i*j + "\t");
   System.out.println();
}
```

מתויגים break -ו continue

- קיימת צורה נוספת של break ו- continue שכוללת תווית, ומאפשרת לצאת ממבנה כלשהו, לאו דווקא המבנה העוטף הסמוך
- הדבר שימושי ביותר ביציאה מלולאות מקוננות
- פרטים בקובץ ההערות שנמצא באתר הקורס

לולאות ומערכים

- לולאות שימושיות ביותר בעבודה עם מערכים
- להלן קטע קוד שמחשב את סכום אברי המערך **arr**:

```
double [] arr = {1.0, 2.0, 3.0, 4.0, 5.0};
double sum = 0.0;
for (int i=0; i<arr.length; i++){
   sum += arr[i];
}
System.out.println("arr sum is: " + sum);
```

- הגדרת משתנה עזר שיהיה אינדקס המערך ואתחולו לאפס
- בדיקת תנאי עצירה – האם משתנה העזר עדיין קטן מגודל המערך
- קידום משתנה העזר באחד
- גוף הלולאה - ביצוע פעולה המשתמשת באיבר במקום ה- **i**

foreach

- ביצוע פעולה מסוימת על כל אברי מערך היא פעולה שכיחה כל כך עד שב-Java5 ניתן לה תחביר מיוחד המכונה foreach (או for/in)
- בתחביר זה הקומפיילר מייצר את העבודה עם משתנה העזר בצורה אוטומטית מאחורי הקלעים
- קטעי הקוד הבאים שקולים:

```
for (int i=0; i<arr.length; i++){
   sum += arr[i];
}
```

שקול ל-

```
for (double d : arr) {
   sum += d;
}
```

קרא זאת כך: "לכל איבר **d** מטיפוס **double** שבמערך **arr** בצע..."

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: " + sumOfPrimes);

    }
}
```

משתנה העזר sumOfPrimes מיותר,
ניתן לוותר עליו

```
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 הם שונות בטיפוס ו\או מספר 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
- 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) שאינן מאבדות מידע
- 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

- נתאר מצב שבו פונקציה המקבלת מספר ארגומנטים, מרבה לקבל את אותם ערכים עבור חלק מהארגומנטים

- כדי לחסוך ממי שקורא לפונקציה את הצורך לציין את כל הארגומנטים (אולי הם רבים, אולי ערכיהם איזוטרים), נעמיס גרסה שבה יש לציין רק את הארגומנטים "החשובים באמת"

העמסה וערכי ברירת מחדל לארגומנטים

- לדוגמא:
  - פונקציה לביצוע ניסוי פיזיקלי מורכב:

```
public static void doExperiment(double altitude,
                double presure, double volume, double mass){
   ...
}
```

  - אם ברוב הניסויים המתבצעים הגובה, הלחץ והנפח אינם משתנים, יהיה זה מסורבל לציין אותם בכל קריאה לפונקציה

  - לשם כך, נגדיר עוד גרסה יותר קומפקטית המקבלת כארגומנט רק את המסה:

```
public static void doExperiment(double mass){
   ...
}
```

העמסה ותאימות לאחור

- נניח כי כתבת את הפונקציה השימושית :

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

המבצעת חישוב כלשהו על הארגומנט **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 מכמה סיבות שנציין מיד

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

- ב Java5 התווסף תחביר להגדרת שרות עם **מספר לא ידוע** של ארגומנטים (vararg)
- 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

- 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)
- משתנים אלו מכונים גם:
  - משתני מחלקה (class variables)
  - אברי מחלקה (class members)
  - שדות מחלקה (class fields)
  - מאפייני מחלקה (class properties/attributes)
  - שדות סטטיים (static fields/members)

- משתנים אלו יוגדרו בתוך גוף המחלקה אך מחוץ לגוף של מתודה כלשהי, וסומנו ע"י המציין **`static`**

משתני מחלקה לעומת משתנים מקומיים

- משתנים אלו, שונים ממשתנים מקומיים בכמה מאפיינים:
  - **תחום הכרות:** מוכרים בכל הקוד (ולא רק מתוך פונקציה מסוימת)
  - **משך קיום:** אותו עותק של משתנה נוצר בזמן טעינת הקוד לזיכרון ונשאר קיים בזיכרון התוכנית כל עוד המחלקה בשימוש
  - **אתחול:** משתנים גלובליים מאותחלים בעת יצירתם. אם המתכנת לא הגדיר 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 +
                         "\tconter 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 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;
```