סטנדרטי פיתוח

בשפת Java

מבוא

כידוע, בכל תהליך פיתוח יש להגדיר סטנדרטים במטרה להקנות הרגלי עבודה נכונים יותר שיעזרו להבנת הקוד ולשמירה על אחידות, בכדי לאפשר עבודת צוות נוחה יותר ומהירה יותר.

חוברת זו מגדירה את הסטנדרטים על פיהם נפתח ונכתוב קוד בשפת Java.

שימו לב כי סטנדרטים אלה מסתמכים על סטנדרטי הפיתוח שgoogle מגדירים. ייתכן ובעת קריאת

חוברת זו תתקלו בנושאים שטרם למדתם, ולכן יש להשתמש בחוברת זו במקביל לתהליך הלמידה שלכם בנושאים השונים, ובכל פעם להתרכז בסטנדרטים הרלוונטיים.

זכרו! הקוד שלכם מייצג אתכם ומעיד על רמתכם המקצועית.

קוד סטנדרטי הוא קוד קריא, ברור ויפה יותר, ולכן ייצג אתכם ברמה ובצורה טובה יותר.

סטנדרטי פיתוח של Google

אנו נשתמש בסטנדרטים המוגדרים של google – Google Java Style Guide.

ניתן להגיע לסטנדרטים בחיפוש בGoogle או דרך הקישור בלחיצה כאן.

שימו לב שבחוברת יהיו דגשים שקיימים באתר.

Google-Java-Format

כדי להתעסק בעיקר, ולא להתעסק בעניין של נראות הקוד מבחינת עימודים, שורות רווח וכדומה, נשתמש בFormatter[1] הנועד לעזור לנו לעמד את הקוד שלנו על פי סטנדרטי הפיתוח של google.

איך מוסיפים?

הכול מוסבר בGitHub בקישור בלחיצה כאן.

שימו לב שמימוש הFormatter הוא open-sourceי וממש אפשר לראות את המימוש של הFormatter.

סיכום שלבי הוספה בIntelliJ

הוספתה Plugin לסביבת העבודה

1. כניסה לSettings)ctrl alt s(
2. כניסה לPlugins
3. חיפוש google-java-format
4. לחיצה על install

כרגע הPlugin התווסף, אך הוא מוגדר כdisabled by default.

כדי לאפשר את פעולותיו בפרויקט הנוכחי:

1. גשו לSettings File→Settings...→google-java-format
2. סמנו v בcheckbox של Enable google-java-format

כדי לאפשר את פעולותיו בפרויקטים חדשים:

1. גשו לSettings File→Other Settings→Settings for New Projects..
2. גשו לgoogle-java-format
3. סמנו v בcheckbox של Enable google-java-format

מה שהתבצע עכשיו זה ששיניתם את פעולת הReformat code,
כאשר נבצע את קיצורי המקשים Ctrl + ALT + L – תוכלו לראות את תוצר הפירמוט של הקוד שלכם על פי הסטנדרטים של Google.

[1] Formatter - תוכנה שמסדרת את נראות הקוד על פי כללים מסוימים

דגש

הFormatter לא עושה לכם סטנדרטים לקוד שלכם – הוא רק משנה את הנראות שלו)עימודים, הזחות...(.

הFormatter עושה סטנדרטים חלקיים.

אין להסתמך אך ורק עליו.

שמות

נקפיד על שימוש בשמות משמעותיים וקריאים – שמות שיכתבו בצורה כזאת יהוו תיעוד של הקוד, יהפכו אותו לקריא יותר ויקלו על הבנתו. השמות בהם נשתמש צריכים להסביר את הקוד לקורא.

כללי

קיימים 2 צורות כתיבה אפשריות לשמות, צורות אלה ישמשו אותנו בהמשך החוברת לכתיבת שמות במצבים שונים:

- lowerCamelCase – שמות שייכתבו בצורה זו ייכתבו באותיות קטנות, כאשר האות הראשונה בכל מילה, מלבד המילה הראשונה, תהיה אות גדולה.
  לדוגמה:

```
sendAllMessages();
```

- UpperCamelCase – שמות שייכתבו בצורה זו ייכתבו באותיות קטנות, כאשר האות הראשונה בכל מילה, כולל המילה הראשונה, תהיה אות גדולה.
  לדוגמה:

```
MessageService { … }
```

משתנים ופרמטרים

- עבור כל משתני מחלקה)משתנה רגיל/קבוע(יש להגדיר modifier – private/public)הכוונה לData Members של המחלקה(
- אין לרשום קדימות שהיא תחילית עבור טיפוסים)לעומת הלימוד בעת"פ(.
- שמות משתנים ייכתבו בlowerCamelCase

```
int countDoctorVisits;
```

- אין לרשום תחיליות שמציינות האם הפרמטר הוא in/out/in+out

קבועים

- קבוע ייחשב רק מה שמוגדר final
  - בשגרות – קבועים יוגדרו כfinal
  - במחלקות – קבועים יוגדרו כstatic final
- שמות קבועים ייכתבו באותיות גדולות עם קו תחתון שמפריד בין המילים

```
Static final int EXIT_CODE = -999;
```

מתודות

- שמות מתודות ייכתבו בצורת lowerCamelCase
- שמות מתודות יתחילו בפעולה ולא בשם עצם

```
public void sendAllMessages() {
    …
}
```

מחלקות

- שמות מחלקות ייכתבו בצורת UpperCamelCase
- שם מחלקה יהיה שם עצם

```
class ExampleClassName {
    …
}
```

Packages

- שמות packageים ייכתבו בצורת lowerCamelCase
- במידה ויש לנו רק package אחד בפרויקט – שמו יהיה כשם הפרויקט שיצרנו.
- במידה ויש יותר מאחד, שמותיהם יהיה שמות משמעותיים שמציינים את החלוקה הלוגית בפרויקט.
- יש להקפיד על כך ששם הpackage לא ייקבע לפי שם מחלקה הנמצאת בו.

```
package packageName
```

תקנים חזותיים

שימו לב שרוב התקנים החזותיים המופיעים כאן – ה formatter מטפל – שכן לא בהכול!

כללי

- סדר המתודות – סדר לוגי ולא כרונולוגי
  - המתודה האחרונה שנוסיף היא לא בהכרח תהיה האחרונה בקוד, אלא במיקום הלוגי המתאים.
- במצב של overloading)העמסה(של ctor ים או מתודות עם שם זהה – הctor ים/מתודות יבואו ברצף אחד אחרי השני בלי ctor ים/מתודות אחרות שיפרידו ביניהם.
- שימו לב להשתמש ברצף המקשים Ctrl + ALT + L לצורך שמירה על עימוד ותיחום נכון.

סוגריים

נעבוד בצורת Egyptian Brackets שנהוג לעבוד בה בJava.

- פתיחת הבלוק { תבוא ללא ירידת שורה – באותה השורה, עם תו רווח אחד אחרי המילה השמורה, ואחריה תבוא ירידת שורה.
- סגירת הבלוק } תבוא בשורה חדשה ונפרדת משלה.
- אחרי סגירת בלוק }, תבוא שורת רווח לפני התחלה של קטע קוד חדש.
- בין סגירת בלוק } לסגירת בלוק }, לא תבוא שורת רווח מפרידה.
- אם לאחר סגירת הבלוק באה מילה שמורה שממשיכה את הבלוק הקודם, כמו else if, try catch וכדומה, מילה שמורה זו תבוא לאחר סגירת הבלוק ללא ירידת שורה)היא תהיה באותה השורה עם הסוגריים המסולסלים הסוגרים – וביניהם תו רווח.
- בבלוק ריק, בלי תוכן , הסוגריים המסולסלים יבואו באותה שורה ללא ירידת שורה)לא נגיע למקרה כזה(
- בכל פסוקית)for, if, else, switch וכדומה(נשתמש בסוגריים מסולסלים – גם אם מדובר בסוגריים ריקים או בסוגריים המכילים פקודה אחת בלבד.

```
private void compareNumbers(int firstNumber, int secondNumber) {
  String output = new String();

  // Check if the first is bigger
  if (firstNumber > secondNumber) {
    output = "First is bigger";
  } else if (firstNumber < secondNumber) { // Second is bigger
    output = "Second is bigger";
  } else {
    output = "The number are equals";

    if (true) {
       output += " - COOL";
    } else { }
  }
}

private void compareNumbers(int firstNumber, int secondNumber) {

  // Calculates sum and returns it
  return (firstNumber + secondNumber) / 2;
}
```

פתיחת הבלוק { באה ללא ירידת שורה ועם תו רווח ישר אחרי סגירת הסוגריים

סגירת בלוק בשורה נפרדת

אין ירידת שורה בין סגירות בלוקים אחד אחרי השני

בלוק ריק יהיה באותה השורה

ירידת שורה לאחר סגירת בלוק

- בcaseים – לא נקיף בסוגריים מסולסלים את קטע הקוד של הcase)נקפיד על עימוד(
- אין להקיף בסוגריים את ערך הcase

```
case POW: // Good
Case (POW(: // Not Good
```

- לפני break תבוא שורת רווח
  - במידה והcase מכיל רק פקודת break – אין צורך בירידת שורה.

```
switch (actionToPerform) {
  case POW:
    // Code

    break;
  case DIVISION:
    // Code

    break;
  default:
    break;
}
```

- return – במידה וערך הreturn הוא מורכב – נקיף בסוגריים, אם לא, אין צורך.

עימוד ותיחום

- כל בלוק שייפתח יהיה מועמד טאב פנימה ביחד לבלוק בו הוא מוכל.
- כל פקודה תהיה בשורה נפרדת.
- אורך שורה יהיה עד 100 תווים
  - יש לעמד בצורה מתאימה במידה ויש חריגה
  - מצבים בהם לא נפצל את השורה:
    - שורות שלא מאפשרות פיצול (ארוך url)
    - שורות של package
    - שורות של import

כללים לפיצול שורה

- כאשר מדובר באופרטור שהוא לא נקודה, פיצול השורה יהיה לפני האופרטור (האופרטור יהיה בשורה הבאה) (הFormatter מעמד לבד)

```
distanceFromUnit =
    Math.sqrt(
        Math.pow(originLocation.coordX - cuurentLocation.coordX)
        + Math.pow(originLocation.coordY - cuurentLocation.coordY));
```

- כאשר מדובר בנקודה, היא תבוא בשורה חדשה ונפרדת.

```
personsList
     .add(new Person(1))
     .add(new Person(2))
     .add(new Person(3))
     .add(new Person(4))
     .add(new Person(5))
     .add(new Person(6));
```

- פסיק יישאר צמוד למה שבא לפניו.
- במידה ויש לולאת for שחורגת מ100 תווים – נפצל כל חלק מלולאת הfor באותו הקו בדיוק.

```
for (int cityLocation = 0;
     cityLocation < AMOUNT_OF_CITIES;
     cityLocation ++) {

}
```

- ביטויים חשבוניים יועמדו זה מתחת לזה, בהתאם לביטוי בתוכו הם נמצאים.

```
distanceFromUnit =
    Math.sqrt(
        Math.pow(originLocation.coordX - cuurentLocation.coordX)
        + Math.pow(originLocation.coordY - cuurentLocation.coordY));
```

ריווח

תווי רווח

- תהיה הפרדה של תו רווח בין מילים שמורות (if for catch...) לסוגריים שבאים אליהם.
- תהיה הפרדה של תו רווח בין מילים שמורות (else, catch) לסוגריים מסולסלים { שקודמים אותם.

```
תו רווח אחרי פסיק
תו רווח לפני פתיחת בלוק למה שבא לפניו
private void compareNumbers(int firstNumber, int secondNumber) {
  String output = new String();
תו רווח לאחר מילה שמורה
תו רווח לפני ואחרי אופרטור
  // Check if the first is bigger
  if (firstNumber > secondNumber) {
    output = "First is bigger";
  } else if (firstNumber < secondNumber) { // Second is bigger
    output = "Second is bigger";
תו רווח בין סגירת בלוק למילה שמורה
  } else {
    output = "The number are equals";

אין תו רווח לפני נקודה פסיק
    if (true) {
      output += " - COOL";
    } else { }
  }
}
תו רווח אחרי נקודה פסיק
private int calculateAverage(int firstNumber, int secondNumber) {
  for (int index; index < firstNumber; index++) { }
תו בין // לפירוט ההערה
  // Calculates average and returns it
  return ((firstNumber + secondNumber) / 2);
}
```

- לפני כל סוגר מסולסל פותח { יבוא תו רווח, מלבד:
  - בין פתיחת סוגריים רגילה לפתיחת סוגריים מסולסלים, לא יבוא תו רווח, למשל:

```
@SomeAnnotation({2, 3});
```

  - בין 2 פתיחות סוגריים מסולסלים או יותר, לא יבוא תו רווח, למשל:

```
String[][] x = {{"foo"}};
```

- משני הצדדים של אופרטור בינארי/משולש – יבוא תו רווח
  - נכון גם לאופרטורים: &, |, : (למשל בforeach)
- אחרי פסיק, נקודתיים, נקודה פסיק – יבוא תו רווח.
- בשני הצדדים של הערה לוגית // - יבוא תו רווח

- בין סוג ושם משתנה

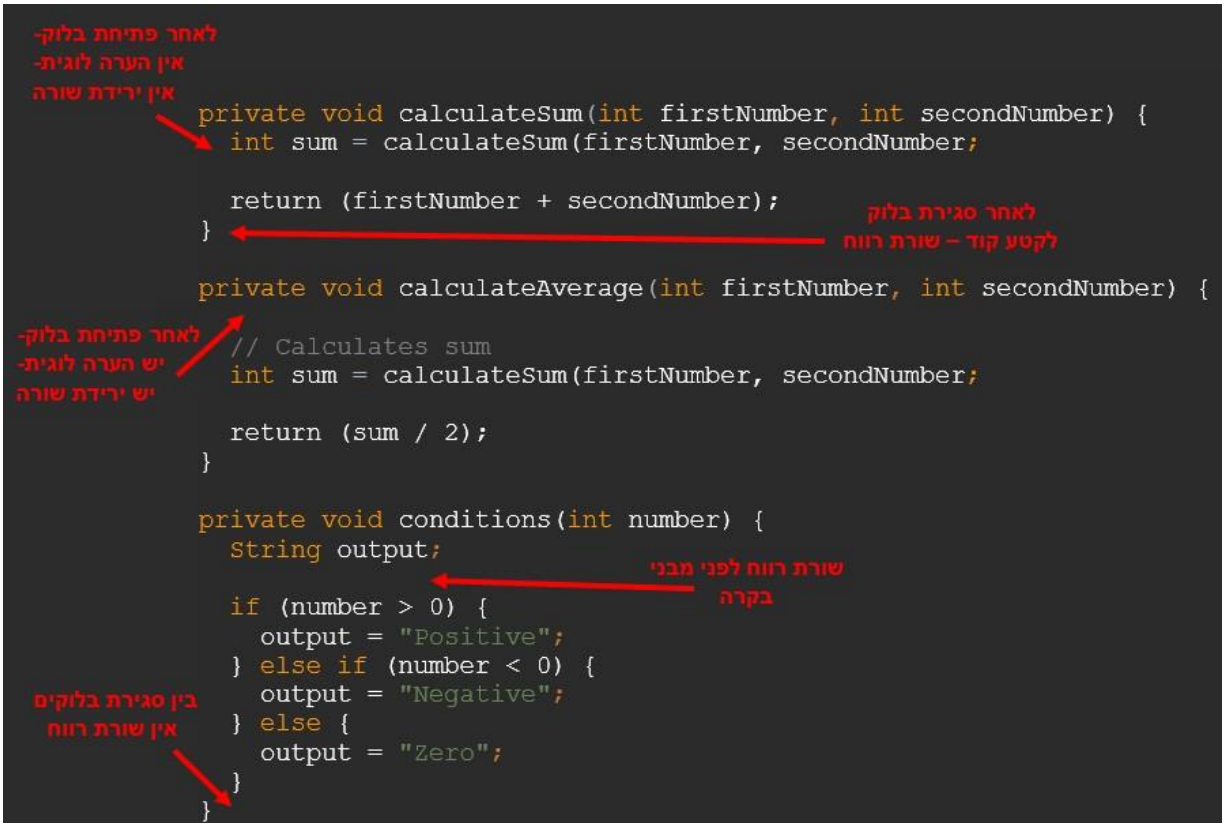
```
int countDoctorVisits;
```

- באתחול של מערך בתוך סוגריים מסולסלים, נשתמש ברווחים באחת משתי הצורות הבאות:

```
new int[] {5, 6}
new int[] { 5, 6 }
```

שורות רווח

- בין שורת הpackage לפקודות הimport תבוא שורת רווח.
- לפני return תבוא שורת רווח
- בפתיחת מתודה, גם main, במידה ויש הערה לוגית, תבוא שורת רווח לפני.
  במידה ואין הערה לוגית, אין לשים שורת רווח בתחילה.
- בין קטע קוד למבנה בקרה, תבוא שורת רווח
- בין סגירת בלוק לסגירת בלוק – לא תבוא שורת רווח
- בין סגירת בלוק לקטע קוד – כן תבוא שורת רווח
- בין מתודות – תבוא שורת רווח
- אין צורך בירידת שורה לאחר הצהרת המחלקה)במידה ותהיה הערה לוגית לאחר מכן, כן נצטרך לרדת שורה(



תיעוד

כללי

אנחנו כותבים את הקוד שלנו בגישה של clean code.

הרבה הערות לא בהכרח גורמות לקוד להיות ברור ומובן יותר, הן יכולות להוסיף "רעש" ולהפוך אותו למסורבל.

במקום לכתוב הערות, נכתוב את הקוד שלנו בצורה טובה וברורה יותר מלכתחילה.

מה הן הערות טובות?

1. Legal comments – זכויות יוצרים
2. Informative comments – הערות המרחיבות את המידע על קטע קוד מסוים)לדוגמה, אזכור של אלגוריתם קיים(3.
Explanation of intent – הסברת הרציונל מאחורי קטע הקוד, אזכור דרישת הלקוח
4. Clarification – הבהרה לגבי משהו שהוא לא בהכרח טבעי בקוד)שימוש במספר קבוע, דילוג על ערכים מסוימים(5.
Warning of consequences – אם אנחנו יודעים שקטע הקוד שלנו יכול ליצור בעיה בעתיד, למשל כאשר משתמשים בספרייה התומכת בגרסה מאוד ספציפית של java.

דגשים כלליים

- הערה באמצעת JavaDoc תעשה על ידי פתיחת סלאש / ולאחריו כתיבת שתי כוכביות ** מעל המחלקה/ממשק/מתודה רצויים.

```
/**
 * Multiple lines of Javadoc text are written here,
 * wrapped normally...
 */
public int method(String parameter) { ... }
```

- או בצורה של שורה אחת כמו בדוגמה הבאה:

```
/** An especially short bit of Javadoc. */
public int method(String parameter) { ... }
```

- בין ההערה לחלק המתואר)מחלקה/ממשק/מתודה(– לא תבוא שורת רווח

הערות לשגרות

- אין צורך בהערות עבור ctor / access methods
- ההערה תכלול הסבר כללי לגבי השגרה
- ההסבר נועד למי שישתמש בשגרה, ולא בהכרח לתוכניתן שיתחזק אח"כ את הקוד שלנו
  - לכן יש להתמקד בממשק והפונקציונליות של השגרה, ולא במימוש.
- לפני ההערה תבוא שורת רווח.

- לאחר ההערה לחתימת השגרה – לא תבוא שורת רווח.
- במידה ויש פרמטרים לשגרה/ערך החזר/זריקת שגיאה – נוסיף הסבר גם לגביהם במיקום המתאים
  - כל אחד מחלקים אלה יופיעו רק במידה ויש חלק קוד תואם לו, במידה ואין זריקת שגיאה אין צורך לרשום את הthrow@ לדוגמה)לא יהיה תיאור ריק(
  - במידה ואחד מהחלקים מכיל יותר משורת הסבר אחת – יש לעמד בהתאם בהזחה של טאב לאחר המילה השמורה

```
/**
 * This method calculate the sum of 2 numbers.
 *
 * @param first - The first number to calculate. This number will always be an
 *     integer. It can be either positive or negative.
 * @param second - The second number
 * @return The sum of the numbers
 * @throws ArithmeticException exception if there is any arithmetic problem
 */
public int calculateSum(int first, int second) throws ArithmeticException {
  return (first + second);
}
```

הערות לוגיות

- יש להסביר חלקים לוגיים בתוכניות או פקודות מסובכות ולא שגרתיות.
- ניתן לכתוב הערה אחת למספר שורות המבצעות מטלה אחת משותפת.
- ניתן)ואף רצוי(להשמיט הערות לפקודות פשוטות במיוחד שניתן להבין בקלות גם ללא הערה.
- הערות ייכתבו באנגלית.
- כל ההערות יתחילו בתו רווח ולאחריו אות גדולה

משתנים, טיפוסים ומבני בקרה

משתנים

- כל משתנה יוגדר בהגדרה נפרדת, בשורה משלו.
- יש להגדיר משתנים מקומיים קרוב ככל הניתן לשימוש הראשון שלו ולא בתחילת הבלוק.
)כלומר, אין יותר אזורי הגדרה של Variable definition.. Code section(
  - לכן, מונה הבקרה של הלולאה, יוגדר בתוך הfor

```
for (int cityLocation = 0;
     cityLocation < AMOUNT_OF_CITIES;
     cityLocation ++) {

}
```

- משתנה מקומי יאותחל מוקדם ככל האפשר)במידה וניתן נאתחל את המשתנה בהגדרה(

ENUM

- אין לשים סוגריים סביב enum בswitch
- ניתן להגדיר enum בשורה אחת וניתן גם להפריד למספרי שורות)נרד שורה לאחר כל פסיק(

מערכים

- בהגדרת מערך, הסוגריים המרובעים יבואו מיד לאחר הטיפוס ולא אחרי שם המשתנה:

```
int arguments[]; // Bad
int[] arguments; // Good
```

- ניתן לאתחל מערכים באחת משתי הצורות הבאות
  1. כל האיברים שבבלוק האתחול של המערך יהיו בשורה אחת
  2. כל איבר בבלוק יהיה בשורה נפרדת

```
int[] numbers = {1, 2, 3};
int[] numbers = {1,
                 2,
                 3};
```

- ניתן להסתמך על האתחול הדיפולטי של התאים במערך, אין צורך לאפס בעצמנו.
- ניתן לבצע מספר פעמים new לאותו המערך. נעשה זאת רק כאשר המערך

  החדש הוא בעל אותה משמעות לוגית כמו המערך הקודם. אנו לא נשתמש באותו מערך על מנת לשמור סוגים שנים של נתונים שאין קשר לוגי בינם.

מתי נשתמש בCollection על פני מערך?

- כאשר רוצים לשמור במערך מספר ערכים לא ידוע מראש
- כאשר רוצים שהערך שאנו שומרים יהיה מקושר למפתח כלשוה
- סוג הCollection ישתנה בהתאם לצורך ולשימוש שאנו רוצים לעשות בו.

Switch

- כל בלוק case יועמד בהזחה פנימה לעומת בלוק הswitch
- אין להשתמש בסוגריים מסולסלים עבור בלוקי case
- כל switch יכיל קטע default – גם אם הוא ריק
- אין צורך בהערה עבור קטע הdefault
- לפני הערות לוגיות המפרטות caseים – תבוא שורת רווח.

Annotations

- אנוטציה של מחלקה/מתודה/CTOR יופיע מיד לפני החתימה
- כאשר למחלקה/מתודה/CTOR יש מספר אנוטציות – הם יופיעו בשורות נפרדות ובאותו הקו

```
@Override
@Nullable
public String name() {…}
```

- כאשר יש רק אנוטציה אחת – ניתן לכתובה באותה השורה עם מי שהוא מוגדר עליו, לדוגמה

```
@Override public String name() {…}
```

Modifiers

עבור כל Data Member יש להגדיר modifier

(public protected private abstract static final transient volatile synchronized native strictfp)

Numeric Literal

ערך מסוג long יסתיים בL גדולה ולא קטנה, כדי למנוע בלבול עם הספרה 1. למשל: 30000L.

חלוקה לאיזורים

אנו נחלק את הקוד שלנו לאזורים כדי שיהיה קריא ונוח לשימוש.

האזורים:

1. Const Members
2. Data Members
3. Ctor
4. Methods

לא נגדיר אזורי הגדרה מפורשים בהערות

מספר דגשים

- לא נגדיר אזור הגדרה בעבור קבועי המחלקה/DMים שלה, שכן נפריד בשורת רווח בין האזורים, ניתן לראות בדוגמה המסכמת בסוף החוברת.

דגשים נוספים

לולאות

- לא נשנה את מונה לולאת מונה הבקרה בתוך הקוד
  - לא נשתמש במונה הלולאה מחוצה לה
  - לא נשתמש באותו משתנה ל2 תפקידים שונים
  - בריצה על collection נשתמש בלולאת foreach

ירושה

- כל מתודה תסומן כOverride@ כאשר זה ניתן.
- במקרים בהם מתודה האב מוגדרת כDeprecated@, לא תסומן המתודה הנוכחית כOverride@.

טיפול בשגיאות

- כמעט תמיד נטפל בשגיאות שתפסנו.
- במקרה ובחרנו לא לטפל כלל בשגיאות שנתפסו – נסביר זאת בהערה ברורה בפסוקית הcatch

מחלקות)class(

- כל class שנכתוב יופיע בקובץ java נפרד
- Reuse – תמיד נעדיף להשתמש בקטעי קוד שכבר כתבנו ולא לכתוב שוב הכול מחדש.
- בכל הפניות למחלקה/דברים מוגדרים בה – נשתמש בthis.
- נגדיר עבור כל מתודה שיש במחלקה את הModifier שלה:
  - Public – ניתנת לגישה מכל מחלקה
  - Private – ניתנת לגישה רק בתוך המחלקה
  - Protected – ניתנת לגישה בעץ הירושה

ממשקים)Interface(

- שם הממשק יהיה בהתאם לייעודו, האות הראשונה שלו תהיה גדולה
- בתוך הממשק נכתוב את הצהרת השגרות ללא מימוש

```java
package example;

import java.util.*;

public class Fighter extends Person {
  private static final String KICK_MESSAGE = "You've been kicked!";

  private Weapon weapon;

  public Fighter(Weapon weapon) {
    super();
    this.changeWeapon(weapon);
  }

  public Weapon weapon() {
    return (this.weapon);
  }

  private void changeWeapon(Weapon weaponToChangeTo) {
    this.weapon = weaponToChangeTo;
  }

  /**
   * Kicks someone
   *
   * @param kickedPerson - person we want to kick.
   * @throws MessageException when the message is invalid.
   */
  public void kick(Person kickedPerson) throws MessageException {
    MessageService.sendMessage(kickedPerson, KICK_MESSAGE);
  }
}
```