ירושה

אחד העקרונות של תיכנות מונחה עצמים הוא, שהתוכנה אמורה לשקף את המציאות - כל עצם בתוכנה משקף עצם כלשהו במציאות.

העצמים במציאות מחולקים לסוגים ותת-סוגים. למשל, בחברה יש עובדים, חלק מהעובדים הם מנהלים, חלק מהעובדים הם מתכנתים, וחלק עובדים פשוטים. אנחנו יכולים לשקף עובדה זו בתוכנה שלנו על-ידי מנגנון ה**ירושה** (inheritance), שנקרא בג'אבה **הרחבה** (extension). מילת המפתח המשמשת להגדרת הרחבה היא **extends**. למשל, נניח שלעובד יש שם ומשכורת, והוא מיוצג ע"י המחלקה:

```
public class Employee {
    String name;
    double salary;
    public String getName() { return name;   }
    public double getSalary() { return salary; }
    ...
}
```

אנחנו יכולים לייצג את העובדה שמנהל הוא סוג של עובד, ע"י ההגדרה הבאה:

```
public class Manager extends Employee {
    ... שדות נוספים ...
    ... מתודות נוספות או מוחלפות ...
}
```

המחלקה המורישה (עובד) נקראת גם super-type, והמחלקה היורשת (מנהל) נקראת גם sub-type.

כיוון שמנהל הוא סוג של עובד, למנהל ישנן כל התכונות של עובד - שם ומשכורת - וגם כל המתודות של עובד. אז אם יש לנו משתנה מסוג Manager:

```
Manager m = new Manager(...)
```

אפשר לגשת למתודה: m.getName() שלו למרות שהיא לא מוגדרת ישירות במחלקה Manager; המחלקה Manager "יורשת" את כל השדות והמתודות מהמחלקה Employee.

בנוסף, כיוון שכל מנהל הוא עובד, ניתן להציב עצם מסוג Manager במשתנה מסוג Employee:

```
Employee e = new Manager(...)
```

אבל ההיפך לא נכון - לא כל עובד הוא מנהל, ולכן לא ניתן להציב עצם מסוג Employee במשתנה מסוג Manager - זו תהיה שגיאת קימפול.

המחלקה היורשת יכולה להוסיף שדות ומתודות חדשים, למשל במחלקה "מנהל" אפשר להוסיף שדה "בונוס" (בהנחה שרק מנהלים מקבלים בונוס):

```
public class Manager extends Employee {
    double bonus;
    public void setBonus(double bonus) {
        this.bonus = bonus;
    }
}
```

(דוגמה נוספת נמצאת במחלקה Programmer - מתכנת. למתכנת יש משתנה נוסף הסופר את מספר שורות הקוד שכתב. הוא מפסיד שקל על כל 10 שורות קוד כדי שיתרגל לכתוב קוד קצר...)

עכשיו, כל משתנה מסוג Manager יכול לקרוא למתודה החדשה: m.setBonus(10).
אבל משתנה מסוג Employee לא יכול לקרוא למתודה החדשה, גם אם יש בו עצם מסוג Manager.
למשל, הקריאה הבאה אינה עוברת קימפול: e.setBonus(10).

בניית מחלקות יורשות

בדרך-כלל, בנאי של מחלקה יורשת קורא לבנאי המתאים של המחלקה המורישה. קריאה זו נעשית ע"י מילת-המפתח super, והיא צריכה להיות בשורה הראשונה של הבנאי. למשל, אם למחלקה "עובד" יש בנאי המקבל שם ומשכורת, אז גם למחלקה "מנהל" בדרך-כלל יהיה בנאי המקבל שם ומשכורת, והוא יקרא לבנאי המתאים של "עובד":

```
public Manager(String name, double salary) {
    super(name, salary);
    bonus = 0;
}
```

אם הבנאי של המחלקה היורשת לא קורא לשום בנאי של המחלקה המורישה, אז הקומפיילר מנסה לקרוא לבנאי-בלי-פרמטרים של המחלקה המורישה. אם אין בנאי כזה, מתקבלת שגיאה. זה יכול לגרום להפתעות. חשבו למשל על התרחיש הבא:

1. מגדירים מחלקה "עובד" בלי שום בנאי.
2. מגדירים מחלקה "מנהל" עם בנאי כלשהו שלא משתמש ב-super. הכל עובד כמו שצריך.
3. מוסיפים למחלקה "עובד" בנאי עם שני פרמטרים. פתאום מקבלים שגיאת קימפול ב"מנהל"!

מדוע? -- כי כשהוספנו בנאי עם פרמטרים למחלקה "עובד", הקומפיילר מחק את הבנאי בלי הפרמטרים.

החלפת מתודות - overriding

מחלקה יורשת יכולה לא רק להוסיף מתודות חדשות - היא יכולה גם **להחליף** מתודות קיימות. החלפת מתודות נקראת באנגלית overriding והיא מתבצעת כך:

```
public class Manager extends Employee {
    ...
    @Override public double getSalary() {
        return super.getSalary() + bonus;
    }
}
```

המתודה getSalary של מנהל מחליפה את זו של עובד פשוט: אצל עובד פשוט, השכר הוא רק שכר הבסיס, בעוד שאצל מנהל, השכר כולל גם בונוס.

שימו לב, המתודה המחליפה משתמשת במילת-המפתח super על-מנת לגשת למתודה של המחלקה המורישה: הקריאה super.getSalary() מחזירה את השכר כפי שהוא מחושב ע"י המתודה getSalary של המחלקה Employee.
מתודה מחליפה לא חייבת לקרוא למתודה במחלקה המורישה, אבל בדרך-כלל היא עושה זאת על-מנת לוודא שהפונקציונליות של המחלקה המורישה נשמרת.

אמרנו קודם שאפשר לשים במשתנה מסוג "עובד" עצם מסוג "מנהל". מה יעשה המחשב במצב זה - לאיזו מתודה getSalary הוא יקרא - של המשתנה או של העצם? בשפת ג'אבה התשובה היא תמיד: **למתודה של העצם**. כלומר כשכותבים e.getSalary() השכר מחושב ע"פ נוסחת השכר של מנהל (כולל הבונוס). שיטת-עבודה זו נקראת late binding - קשירה מאוחרת. קשירת המתודה לקוד מתבצעת בשלב מאוחר - בזמן ריצת התוכנית (ולא בזמן הקומפילציה).

אגב, התיוג @Override אינו חובה. הקוד יעבוד בדיוק באותה מידה גם אם נשמיט אותו. מטרת התיוג הזה היא להגיד לקומפיילר "היי, שים לב, אנחנו רוצים להחליף מתודה שנמצאת במחלקה המורישה". ולמה זה עוזר לנו? כי אם, למשל, נעשה שגיאת כתיב ונכתוב בטעות "getSlaary", אז בלי התיוג Override הקומפיילר יניח שאנחנו רוצים להוסיף מתודה חדשה, ועם התיוג Override הקומפיילר יתריע על שגיאה ואנחנו נוכל לתקן אותה לפני שהיא תגרום נזק.

שיטת "הקשירה המאוחרת" מאפשר לבצע חישובים מורכבים בקלות. למשל, אפשר להחזיק מערך או רשימה של עובדים:

```
List<Employee> employees = new ArrayList<>();
```

להכניס לתוכה עובדים מסוגים שונים, ואז לחשב בבת-אחת את השכר הכולל שצריך לשלם לכל העובדים בחברה:

```
double totalSalary =
     employees
     .stream()
     .mapToDouble(e -> e.getSalary())
     .reduce( (x,y) -> x+y )
     .getAsDouble();
```

ראו במחלקה EmployerDemoOverriding.

תרגיל כיתה: נניח שאתם משתמשים ב-ArrayList ומגלים שגיאה - המערך שלכם מכיל פריט מוזרים שאתם לא מבינים מאיפה הגיעו לשם. אתם רוצים לכתוב הודעה למסך בכל פעם שמישהו מוסיף פריט למערך. כיתבו מחלקה היורשת את ArrayList ומחליפה את המתודות המתאימות. כיתבו מחלקה זו בשתי דרכים: כמחלקה עם שם וכמחלקה אנונימית:

```
List<String> = new ArrayList<String>() { ... }
```

(הפתרון: בקבצים ArrayListWithLogging, ArrayListWithLoggingDemo).

העמסת מתודות - overloading

ישנה שיטה נוספת להגדיר מתודות עם שם זהה ומימוש שונה:

```
private static double getSalary(Employee e) {
     return e.salary;
}
private static double getSalary(Manager m) {
     return m.salary + m.bonus;
}
```

שיטה זו נקראת **העמסה** - overloading.

גם במצב זה, אם נקרא ל-getSalary(e) נקבל שכר של עובד פשוט, ואם נקרא ל-getSalary(m) נקבל שכר של מנהל. אבל מה יקרה אם נחשב את השכר הכולל ברשימה של עובדים?

```
double totalSalary =
     employees
     .stream()
     .mapToDouble(e -> getSalary(e))
     .reduce((x,y)->x+y)
```

```
.getAsDouble();
```

הפתעה! המחשב התייחס רק לשכר-הבסיס והתעלם מהבונוסים (ראו במחלקה EmployerDemoOverloading).

מדוע? -- כי בהעמסה, מתבצע **קשירה מוקדמת** (early binding) - קשירת המתודה לקוד מתבצעת בשלב מוקדם - בשלב הקומפילציה. כיוון שכל האיברים ברשימה הם משתנים מסוג Employee, הקומפיילר קורא למתודה getSalary שמקבלת כקלט משתנה מסוג Employee, והמתודה הזאת מתעלמת מהבונוס.

ריבוי צורה - Polymorphism

המושג **ריבוי צורה** מציין את האפשרות לבצע פעולה אחת (כגון חישוב משכורת) בכמה צורות שונות אצל עצמים שונים. כפי שראינו, בג'אבה ישנם שני סוגים של ריבוי צורה:

- החלפה - overriding - שבה הקשירה היא מאוחרת;
- העמסה - overloading - שבה הקשירה היא מוקדמת.

שימו לב לסכנה נוספת הנובעת מערבוב בין החלפה לבין העמסה. נניח שלעובד יש מתודה הבודקת האם הוא בדרגה נמוכה יותר מעובד אחר (נניח שזה מחושב לפי ותק בחברה):

```
public boolean lowerRankThan(Employee other) {
     return this.joinDate.isAfter(other.joinDate);
}
```

במחלקה מנהל, אנחנו רוצים להחליף את המתודה הזאת במתודה הבודקת האם הוא בדרגה נמוכה יותר ממנהל אחר (נניח שבין מנהלים זה מחושב לפי הבונוס; ברור שמנהל הוא לא בדרגה נמוכה יותר מעובד פשוט). אז אנחנו כותבים:

```
public boolean lowerRankThan(Manager other) {
     return this.bonus < otherManager.bonus;
}
```

להפתעתנו, מתברר שלא **החלפנו** את המתודה אלא **העמסנו** אותה. עכשיו למחלקה Manager יש שתי מתודות שונות בשם lowerRankThan; הקומפיילר יחליט לאיזה מהן לקרוא. בפרט, אם נקרא למתודה עם משתנה מסוג Employee, תיקרא המתודה של Employee ולא של המחלקה החדשה!

מדוע? -- כי בשפת ג'אבה, מתודה מוגדרת לא רק לפי השם שלה אלא גם לפי הארגומנטים שהיא מקבלת. שתי מתודות עם שם זהה וארגומנטים שונים נחשבות לשתי מתודות שונות. לכן, השניה לא מחליפה את הראשונה אלא רק מעמיסה עליה.

כדי להימנע מטעויות כאלו, כדאי שוב להשתמש בתג @Override. אם נכתוב:

```
@Override public boolean lowerRankThan(Manager other) { ... }
```

נקבל שגיאת קימפול ונבין שטעינו.

אם בכל-זאת רוצים להחליף את המתודה, צריך שהארגומנטים יהיו זהים. אבל אז מתעוררת בעיה אחרת: איך נוכל לגשת לשדה הבונוס של other? הפתרון הוא להשתמש ב- **instanceof** ובהשלכה (casting):

```
@Override public boolean lowerRankThan(Employee other) {
```

```
    if (other instanceof Manager) {
        Manager otherManager = (Manager)other; // casting
        return this.bonus < otherManager.bonus;
    } else {
        return false;
    }
}
```

האופרטור **instanceof** בודק האם עצם מסויים שייך למחלקה מסויימת היורשת את המחלקה שבה נמצא המשתנה. אם התשובה היא כן, אז ניתן **להשליך** את העצם למחלקה הזאת. השלכה נקראת באנגלית casting והיא מתבצעת ע"י כתיבת שם המחלקה היורשת בסוגריים לפני העצם; ראו למעלה.
חידה: מה יקרה אם ננסה לבצע את ההשלכה בשורה הראשונה של המתודה, לפני שהשתמשנו באופרטור instanceof?

שימו לב 2: הערך המוחזר ממתודה **אינו** חלק מהגדרת המתודה. לכן, אם הגדרנו במחלקה "עובד" מתודה שמחזירה "עובד":

```
public Employee getSupervisor() { ... }
```

אפשר להחליף אותה במחלקה "מתכנת" ע"י מתודה שמחזירה "מנהל":

```
@Override public Manager getSupervisor() { ... }
```

מדוע? כי המתודה המוחלפת מתחייבת להחזיר "עובד", והמתודה המחליפה מחזקת את ההתחייבות ומתחייבת להחזיר מנהל, שהוא סוג של עובד. כל עוד הערך המוחזר במתודה המחליפה הוא זהה או תת-סוג של הסוג המוחזר במתודה המוחלפת, ההחלפה תקינה.
חידה: מה יקרה אם המתודה המחליפה תנסה להחזיר String?

מילות-מפתח להגבלת הירושה

ישנן שתי מילות-מפתח שמתכנת יכול להשתמש בהן על-מנת להגביל את האפשרות של מתכנתים אחרים להשתמש במתודות ובמחלקות שיצר. הסיבה העיקרית להשתמש במילים אלו היא "לפני עיוור לא תתן מכשול" - המתכנת רוצה לחסוך למתכנתים הבאים אחריו שגיאות שעלולות לנבוע משימוש לא נכון במחלקה שלו.

1. **סופי - final**: שימוש במילה זו לפני הגדרת מחלקה (לפני המילה class) מגדיר את המחלקה כ"סופית" ולא מאפשר כלל להגדיר לה מחלקות יורשות (ניסיון לרשת אותה יגרום לשגיאת קימפול). כמו כן, שימוש במילת-מפתח זו לפני שם של מתודה, מגדיר את המתודה כ"סופית" ולא מאפשר למחלקות יורשות להחליף אותה (ניסיון להחליף אותה יגרום לשגיאת קימפול). המילה הזאת מעבירה מסר למתכנתים הבאים: "אל תנסו להחליף/לרשת את המתודה/המחלקה - אין טעם - זה עלול לגרום לכם שגיאות". דוגמה למתודה סופית בג'אבה: getClass. דוגמה למחלקות סופיות בג'אבה: String, URL, LocalTime.

2. **מופשט - abstract**: שימוש במילה זו לפני הגדרת מחלקה מגדיר את המחלקה כ"מופשטת" ולא מאפשר לבנות עצמים ישירות מהמחלקה הזאת (ניסיון לבנות עצם ממחלקה מופשטת גורם שגיאת קימפול). המילה הזאת מעבירה מסר למתכנתים הבאים: "כל העצמים במערכת שייכים לאחת התת-מחלקות; אין עצם ששייך ישירות למחלקת-הבסיס". למשל, אם אנחנו יודעים שכל העובדים בחברה הם מנהלים או מתכנתים, אז אנחנו יכולים להעביר ידע זה למתכנתים הבאים אחרינו ע"י הגדרת המחלקה Employee כמופשטת. במחלקה מופשטת אפשר גם להגדיר מתודות מופשטות ע"י כתיבת המילה abstract לפני שם המתודה; במקרה זה, למתודה אין גוף (אין מימוש), וכל מחלקה לא-מופשטת שתירש מהמחלקה המופשטת שלנו, תהיה חייבת לספק מימוש למתודה זו.

הערה: ישנו דמיון רק בין מחלקה מופשטת לבין ממשק (interface). ההבדל העיקרי הוא, שלמחלקה מופשטת יכולים להיות שדות, ולממשק אין שדות.

המילים final ו-abstract מנוגדות - הראשונה לא מאפשרת ירושה, השניה מחייבת ירושה. לכן כמובן אי-אפשר להשתמש בשתיהן בו-זמנית.

הגבלת גישה לשדות ומתודות

כזכור, המילה private מגבילה את הגישה לשדה/מתודה, כך שיהיה אפשר לגשת אליהם רק מתוך המחלקה.

ללא המילה private, אפשר לגשת לשדה/מתודה מתוך המחלקה וגם מתוך החבילה.

המילה protected מאפשרת לגשת לשדה/מתודה מתוך המחלקה, מתוך החבילה, **וגם מתוך המחלקות היורשות**.

המילה public מאפשרת גישה ציבורית בלי הגבלה.

מתי **לא** להשתמש בירושה?

מתכנתים רבים חושבים שירושה היא בסה"כ מנגנון לחיסכון בכתיבת קוד. זה נכון ששימוש בירושה חוסך קוד: במקום לכתוב מחדש ב-Manager את כל השדות והמתודות שכבר כתבנו ב-Employee, אנחנו יורשים אותם מ-Employee וכותבים רק את השדות והמתודות החדשים.

אבל, המחשבה שירושה היא **רק** מנגנון לחיסכון בקוד היא שגויה ועלולה להביא לתיכנון לא נכון של מחלקות.
לדוגמה, נניח שאנחנו רוצים להגדיר מחסנית. כזכור, מחסנית היא מבנה-נתונים שמאפשר להכניס לתוכו פריטים ולהוציא אותם בסדר הפוך מסדר ההכנסה. איפה נשמור את הפריטים? כנראה ברשימה. מתכנת נאיבי יכול לחשוב, אם כך, שמחסנית צריכה לרשת מרשימה ולהוסיף עליה את המתודות של מחסנית, למשל:

```
public class Stack<T> extends ArrayList<T> {
     public void push (T item) { add(item); }
     public T pop() { return remove(size()-1); }
}
```

כך לכאורה חוסכים קוד, למשל המחסנית שלנו יורשת את המתודה toString ואין צורך להגדירה מחדש.
מה הבעיה? הבעיה היא שמחסנית אינה תת-סוג של רשימה. מחסנית היא מבנה-נתונים שונה: היא מבטיחה שהפריטים ייצאו ממנה בסדר הפוך לסדר הכניסה. רשימה לא מבטיחה את זה. הנה קטע קוד שמדגים את הבעיה:

```
Stack0<Integer> stack0 = new Stack0<>();
stack0.push(111);
stack0.push(222);
stack0.push(333);
stack0.add(1, 444);
System.out.println(stack0.pop());
System.out.println(stack0.pop());
```

```
System.out.println(stack0.pop());
System.out.println(stack0.pop());
```

סדר ההוצאה לא יהיה הפוך מסדר ההכנסה! זו לא מחסנית אמיתית.

במציאות, ירושה לא נכונה יכולה לגרום לבאגים נסתרים וקשים לגילוי. הכלל לירושה נכונה הוא: מחלקה ב יורשת ממחלקה א, רק אם במציאות סוג ב הוא תת-סוג של סוג א.
ניסוח מפורט יותר של כלל זה הוא **עקרון ההחלפה של ליסקוב** (Liskov Substitution Principle): מחלקה ב יורשת ממחלקה א, רק אם בכל מקום שבו משתמשים בעצם ממחלקה א, אפשר להשתמש בעצם ממחלקה ב בלי לקלקל את התוצאות.

אם העקרון לא מתקיים - לא משתמשים בירושה. במה כן משתמשים? ב**הרכבה** (composition).
הרכבה היא, פשוט, הכנסת העצם ממחלקה א כשדה במחלקה ב. למשל, את המחסנית שלנו נגדיר באופן הבא:

```
public class Stack<T> {
     private List<T> list = new ArrayList<>();
     public void push (T item) { list.add(item); }
     public T pop() { return list.remove(list.size()-1); }
     @Override public String toString() { return "Stack:
"+list.toString();}
}
```

למה זה עדיף?

- הקוד ברור יותר - ברור שמחסנית היא מבנה-נתונים עצמאי שמשתמש ברשימה, והיא לא סוג של רשימה.
- עכשיו, כל ניסיון להכניס פריט באמצע המחסנית ייתקל בשגיאת קומפילציה. כך אנחנו חוסכים טעויות למתכנתים שיבואו אחרינו ומקיימים את המצוה "לפני עיוור לא תתן מכשול".

אגב, המתכננים של שפת ג'אבה עצמה עשו את הטעות הזאת והגדירו מחלקה בשם Stack שיורשת מ-Vector... הם התחרטו, והחל מגירסה 6 הם ממליצים לא להשתמש במחלקה זו.

המחלקה Object

בג'אבה ישנה מחלקה בשם Object, שכל המחלקות האחרות יורשות ממנה אוטומטית. למשל, המחלקה Employee למעשה מאחרי הקלעים מוגדרת כך:

```
public class Employee extends Object { ... }
```

למה זה טוב? -- כי במחלקה Object מוגדרות כמה מתודות שכדאי שיהיו לכל אובייקט (כך לפחות חשבו מתכנני השפה לפני 20 שנה).

מתודה אחת שכבר ראינו היא toString המחזירה ייצוג טקסטואלי של העצם.

מתודה נוספת היא getClass - היא מחזירה עצם מסוג Class המייצג את המחלקה של העצם הנוכחי. ניתן להשתמש בעצם זה כדי להדפיס את המחלקה של העצם וכן כדי לבדוק אם שני עצמים שייכים לאותה מחלקה.

מתודות נוספות נראה בהמשך.

בדיקת שיוויון - equals, hashCode, compareTo

במחלקה Object מוגדרת המתודה equals שתפקידה לבדוק האם עצם זה זהה לעצם אחר.
למה צריך את זה?
כי יש הרבה פעולות שמסתמכות על השאלה האם עצמים מסויימים הם זהים או שונים. למשל, אנחנו יכולים לשים עצמים ברשימה list, ואז לבדוק האם עצם כלשהו נמצא ברשימה:
list.contains(o); כדי לבדוק אם העצם o מופיע ברשימה, צריך לעבור על כל העצמים ברשימה ולבדוק אם אחד מהם שווה ל-o. אבל מה זה אומר בדיוק ששני עצמים הם שווים? זה משתנה ממחלקה למחלקה.
ברירת-המחדל המוגדרת במחלקה Object היא, ששני עצמים נחשבים שווים אם הם אותו עצם בדיוק. אבל במקרים רבים הבדיקה הזאת מחמירה מדי, ובמקרים כאלה צריך להחליף (Override) את המתודה equals. למשל, שתי מחרוזות נחשבות לשוות אם יש בהן אותן אותיות - גם אם זה לא אותו עצם בדיוק; לכן המחלקה String מגדירה מחדש את equals.

אם אנחנו רוצים להגדיר מחדש את equals במחלקה שכתבנו, אנחנו צריכים להחליט מה הם השדות החשובים המגדירים שיוויון בין עצמים. לדוגמה, במחלקה Employee, אפשר לקבוע שהשדות החשובים הם שם ומשכורת, או לחלופין מזהה-עובד (אם יש) - תלוי באפליקציה. הדרך הפשוטה ביותר להחליף את המתודה equals היא להשתמש בשירות של אקליפס: Source -> generate toString; השירות הזה עושה עבורנו הרבה פעולות חשובות כמו בדיקת null, בדיקה שהמחלקה היא אותה מחלקה, וכו' - וודאו שאתם מבינים את התפקיד של כל שורה.

שימו לב שאקליפס יוצר אוטומטית גם equals וגם hashCode. מדוע? כי לפי כללי השפה, שתי המתודות הללו חייבות להיות תואמות: אם שני עצמים הם שווים לפי equals, אז הם חייבים להחזיר את אותו hashCode. לכן, שתי המתודות חייבות להתייחס לאותם שדות. אם זה לא המצב (שני עצמים שווים לפי equals אבל מחזירים קוד-עירבול שונה), אז עדיין אפשר לעבוד עם עצמים מהמחלקה הזאת במבני-נתונים כגון ArrayList, אבל אם ננסה לעבוד במבני-נתונים הדורשים קוד-עירבול (HashSet, HashMap), לא נוכל למצוא את הפריטים שהכנסנו לשם; לדוגמה ראו EmployeeDemoEquals.

חידה: מה קורה במצב ההפוך - שני עצמים הם שונים לפי equals אבל מחזירים קוד-עירבול זהה?
תשובה: התוכנית עובדת בצורה תקינה. למעשה המצב חייב להיות כך - יש הרבה יותר עצמים ממספרים שלמים, כך שמשיקולי ספירה, בהכרח יהיו עצמים שונים עם קוד-עירבול זהה. טבלאות-עירבול יודעות להתמודד עם המצב הזה - לפרטים ראו בקורס "מבני נתונים".
אם כך, מדוע לא לכתוב את hashCode כך שתחזיר תמיד 0? --- משיקולי יעילות זמן ריצה. לפרטים ראו בקורס "מבני נתונים".

הערה: יש סוג שלישי של מתודת-השוואה - compareTo - שגם היא צריכה להיות תואמת ל-equals. המתודה הזאת **אינה** מוגדרת במחלקה Object אלא בממשק Comparable, כך שצריך להגדיר אותה רק כשרוצים לממש את הממשק Comparable. משתמשים בה כשרוצים להכניס עצמים למבנים מסויימים כגון TreeSet, ConcurrentSkipListSet.

שיבוט - clone

במחלקה Object מוגדרת המתודה clone. מטרת המתודה היא **לשבט** את העצם - ליצור עצם חדש זהה, שאפשר לשנות אותו בלי לשנות את העצם המקורי. בדרך-כלל, כשמעתיקים משתנים ומשנים אחד מהם, גם השני משתנה, למשל, אחרי הקוד הבא:

```
Employee e1 = new Employee("A",100);
Employee e2 = e1;
e2.setName("B");
```

גם השם של e1 הוא B, כי ההעתקה ע"י סימן "=" היא העתקה של מצביע - אחרי ההעתקה, יש שני משתנים שונים המצביעים לאותו עצם בזיכרון. כדי ליצור העתק של העצם, אפשר להשתמש ב-clone, למשל כך:

```
Employee e2 = e1.clone();
```

אבל, זה לא כל כך פשוט. כדי שהשורה תעבוד, צריך לעשות כמה דברים:

א. להגדיר את העצם Employee כ-Cloneable.

ב. לממש את המתודה clone ב-Employee:

```
public Employee clone() {
        ...
}
```

ג. להגדיר את המתודה clone ב-Employee כציבורית - public (כי במחלקה Object היא מוגדרת כ-protected).

ד. לטפל בחריגה CloneNotSupportedException.

זה מסובך ולא משתמשים בזה הרבה. הדרך הנוחה יותר לשבט עצמים היא ע"י בנאי מעתיק:

```
public Employee(Employee other) {
        ....
}
```

בדרך זו אין צורך לממש את הממשק Cloneable ואין צורך לטפל בחריגה CloneNotSupportedException.

החיסרון של בנאי-מעתיק הוא, שכדי להשתמש בו צריך לדעת את הסוג המדוייק של העצם. אפשר לגלות את הסוג המדוייק ע"י getClass, משם אפשר לקבל בנאי ולהפעיל אותו.

אם בכל-זאת רוצים שיבוט בסגנון של clone שאינו דורש לדעת את סוג העצם, עדיף להשתמש בספריה חיצונית הממשת clone בצורה יעילה יותר, למשל Kryo, Apache Commons, Cloner.

המתנה - wait, notify, notifyAll

במחלקה Object מוגדרת המתודה wait. ניתן להריץ מתודה זו על עצם מסויים רק בתוך בלוק סינכרון על אותו עצם, למשל:

```
synchronized (object) {
     ... object.wait(); ...
}
```

כשתהליך מבצע פעולה זו, הוא משחרר את המנעול על object, ונכנס להמתנה. התהליך יתעורר מהמתנה כשתהליך אחר יבצע את הפעולה המשלימה: **object.notifyAll()**.

(אם כמה תהליכים מחכים על אותו עצם, אז notifyAll מעיר את כולם, ו-notify מעיר רק אחד מהם).

למה צריך את זה? -- למשל כדי לממש מבני-נתונים מסונכרנים מתוחכמים כגון BlockingQueue שלמדנו באחד השיעורים הקודמים. רוב המתכנתים לא יצטרכו להשתמש במתודות האלו באופן ישיר.

מקורות

- Cay Horstmann, “Core Java SE 9 for the Impatient”, chapter 4.
- Cay Horstmann, “Core Java SE 9 for the Impatient”, chapter 10.7.3 "Waiting on conditions".
- Dane Cameron, "A Software Engineer Learns Java and OOP", chapter 12.
- "Clone is broken", Josh Block, http://www.artima.com/intv/bloch13.html
- "Inheritance vs. Composition", JavaWorld https://www.javaworld.com/article/2076814/core-java/inheritance-versus-composition--which-one-should-you-choose-.html
- "Polymorphism vs. overriding vs. overloading", StackOverflow https://stackoverflow.com/q/154577/827927
- "Negative aspects of Java Stack extending Vector", StackOverflow, https://stackoverflow.com/q/2922257/827927
- "Copy constructor vs. clone", StackOverflow, https://stackoverflow.com/a/1106159/827927
- "Deep clone in Java", StackOverflow, https://stackoverflow.com/q/64036/827927

סיכם: אראל סגל-הלוי.