אוספים

אוסף הוא מבנה-נתונים שאפשר להכניס לתוכו פריטים, לבדוק איזה פריטים קיימים בו, ולהוציא מתוכו פריטים. בוודאי שמעתם על סוגים שונים של אוספים, כגון: מערך, תור, עץ בינארי וטבלת גיבוב. יש עוד הרבה סוגים של אוספים עם תכונות שונות. בשפת ג'אבה אפשר לגשת לכולם דרך **ממשק** אחד – הממשק Collection. זוהי דוגמה ליופי בשיטת הממשקים: מגדירים ממשק אחד וממשים אותו בהרבה דרכים שונות. זה ממשק **גנרי** – יש לו פרמטר-סוג המציין את סוג הפריטים שרוצים שיהיו באוסף. למשל כדי להגדיר משתנה שהוא אוסף של מחרוזות אפשר לכתוב:

```
Collection<String> c;
```

ניתן לאתחל משתנה כזה עם כל אחד מהמימושים של Collection – בהמשך נראה כמה מהם. הפשוט ביותר הוא ArrayList. אפשר לאתחל את המשתנה שיצרנו קודם כך:

```
Collection<String> c = new ArrayList<>();
```

שימו לב, כיוון שהגדרנו את המשתנה כאוסף של מחרוזות, הקומפיילר יודע לבד שהמערך שצריך ליצור הוא מערך של מחרוזות.

מתודות של אוספים

המתודות העיקריות של הממשק Collection הן:

(א) מתודות להוספת חפצים – add (הוספת חפץ אחד), addAll (הוספת כל החפצים הנמצאים באוסף אחר). כדי להוסיף הרבה חפצים בבת-אחת בלי ליצור אוסף חדש, אפשר להשתמש במתודה Collections.addAll.

(ב) מתודות להסרת חפצים – remove (הסרת חפץ אחד), removeAll (הסרת כל החפצים הנמצאים באוסף אחר), retainAll (הסרת כל החפצים **שאינם** נמצאים באוסף אחר), clear (הסרת כל החפצים), והכי שווה – removeIf. המתודה האחרונה מקבלת כפרמטר **פרדיקט** (עצם הממש את הממשק Predicate) הקובע איזה חפצים להסיר. לדוגמה:

```
Collections.addAll(c, "bcde","abc","cdefg","de");
c.removeIf(s->s.length()>=4);
System.out.println(c); // prints [abc, del]
```

(ג) שאילתות – size (כמה חפצים יש באוסף), isEmpty (האם הוא ריק), contains (האם הוא מכיל חפץ מסויים), containsAll (האם הוא מכיל את כל החפצים באוסף אחר).

(ד) המרה למערך רגיל. יש שתי מתודות בשם toArray: אחת לא מקבלת שום ארגומנטים ומחזירה מערך של Object – היא לא כל כך נוחה כי נצטרך אחר-כך להמיר כל Object לסוג של העצמים שיש במערך. השימושית יותר מקבלת ארגומנט אחד שהוא מערך מהסוג שאנחנו רוצים ליצור. אם המערך מספיק גדול המתודה תשתמש בו, אחרת המתודה תיצור מערך גדול יותר מאותו סוג. למשל, כדי להעתיק אוסף של מחרוזות למערך נשתמש ב:

```
String[] a = c.toArray(new String[0]);
```

(ה) מעבר על הפריטים במערך – המתודות iterator, stream, parallelStream, spliterator. בשיעור זה נתמקד בפשוט ביותר – **איטרטור** (iterator).

(ו) בנוסף למתודות של Collection, ישנם אלגוריתמים רבים שאפשר לבצע בעזרת מתודות סטטיות של המחלקה **Collections**, למשל: disjoint – בדיקה האם שני אוספים הם זרים; frequency – ספירה כמה פעמים פריט מסויים מופיע באוסף, ועוד. לפרטים ראו בדף הממשק Collections בתיעוד של ג'אבה.

איטרטורים

איטרטור הוא גם-כן ממשק. יש לו שלוש מתודות עיקריות:

- hasNext – האם יש עוד פריטים באוסף?
- next – התקדם והחזר את הפריט הבא באוסף;
- remove – הסר את הפריט האחרון שהוחזר על-ידי next.

לדוגמה, הקטע הבא מסיר את כל המחרוזות שאורכן לפחות 4:

```
for (Iterator<String> i = c.iterator(); i.hasNext(); ) {
     String s = i.next();
     if (s.length()>=4) i.remove();
}
```

במקרה זה כמובן עדיף להשתמש במתודת removeIf שראינו קודם – היא עושה אותו דבר בשורה אחת במקום ארבע.

חידה: מצאו בעיה שכדי לפתור אותה חייבים להשתמש ב-remove ואי אפשר להשתמש ב removeIf.

אם רוצים רק לעבור על הפריטים באוסף בלי להסיר אותם, אפשר להשתמש בתחביר מקוצר:

```
for (String s: c) {
     System.out.println(s);
}
```

מאחרי הקלעים, הקומפיילר החכם של ג'אבה מתרגם את הלולאה הזו ללולאה דומה לקודמת – הוא יוצר איטרטור מהאוסף c, מריץ אותו כל עוד hasNext מחזיר "אמת", ובכל הרצה שם במשתנה s את הערך של next. כיוון שהאיטרטור עצמו נמצא רק מאחרי הקלעים, אי אפשר לבצע remove בתחביר המקוצר.

שימו לב: התחביר המקוצר פועל אוטומטית עבור כל מחלקה שמממשת את הממשק Iterable. הממשק Collection מרחיב (=יורש) את הממשק Iterable ולכן התחביר הזה פועל גם על כל האוספים.

יש תחביר עוד יותר מקוצר למעבר על אוספים – הוא משתמש במתודה forEach המקבלת כקלט עצם הממשק את הממשק Consumer; זה נשמע מסובך אבל למעשה זה פשוט:

```
c.forEach( s -> System.out.println(s) );
```

קבוצות

לממשק Collection ישנם כמה ממשקים שמרחיבים אותו (=יורשים ממנו), ולכל אחד מהם יש כמה מימושים. כדאי להכיר אותם כדי שנדע לבחור את המימוש המתאים לכל מצב.

הממשק Set מציין **קבוצה** – שבה כל פריט מופיע לכל היותר פעם אחת. גם אם מוסיפים את אותו פריט כמה פעמים, עדיין יישאר רק עותק אחד. המימושים נבדלים ביניהם בסדר שבו הפריטים מופיעים בקבוצה: במימוש HashSet הסדר הוא שרירותי, במימוש LinkedHashSet הסדר הוא אותו הסדר שבו הוכנסו הפריטים לקבוצה, ובמימוש TreeSet הסדר הוא הסדר ה"טבעי" של הפריטים – הסדר

שנקבע ע"י המתודה compareTo של הפריטים או ע"י עצם מסוג Comparator המועבר לבנאי.
לדוגמה, נניח שיש לנו מתודה כזאת:

```
static void testAdd(Collection<Integer> c) {
      Collections.addAll(c, 10, 90, 20, 80, 30, 70, 40, 60, 50, 50);
      System.out.println(c);
}
```

ואנחנו מריצים אותה ארבע פעמים:

```
testPrint(new HashSet<>());
testPrint(new LinkedHashSet<Integer>());
testPrint(new TreeSet<Integer>());
testPrint(new TreeSet<Integer>((x,y)->y-x));
```

בכל הפעמים נקבל את אותם פריטים – המספר 50 יופיע רק פעם אחת למרות שהכנסנו אותו פעמיים.
אבל בפעם הראשונה סדר ההדפסה יהיה שרירותי ולא צפוי מראש, בפעם השניה נראה 10, 90, 20, 80...
לפי סדר ההכנסה, בפעם השלישית נקבל 10, 20, 30... ובפעם הרביעית נקבל סדר הפוך 90, 80, 70...

למחלקה TreeSet יש עוד כמה מתודות שקשורות לסדר – אפשר לקבל את הפריט הראשון בקבוצה לפי הסדר הטבעי (first), האחרון (last), כל הפריטים לפני פריט מסויים (headSet), אחרי פריט מסויים (tailSet), בין שני פריטים (subSet), ועוד; ראו בתיעוד.

למה בכלל צריך את HashSet אם סדר הפריטים לא צפוי מראש? התשובה היא **מהירות**. המבנה HashSet ממומש ע"י טבלת גיבוב (Hash table) המאפשרת הכנסת פריטים ובדיקת הימצאות של פריטים בזמן כמעט קבוע O(1), בעוד שהמבנה TreeSet ממומש ע"י עץ בינארי מאוזן שבו כל פעולה דורשת זמן O(log(n)) כאשר n הוא מספר הפריטים בעץ. המבנה LinkedHashSet ממומש ע"י טבלת גיבוב בתוספת קישורים בין הפריטים, הוא דורש יותר זיכרון מ-HashSet. כדאי להכיר את התכונות של כל מימוש כדי לבחור את המימוש המתאים.

חידה: נתונות שתי קבוצות a,b. איך תחשבו בג'אבה את **האיחוד** שלהן? את **החיתוך** שלהן? את **ההפרש** ביניהן?

רשימות

הממשק List מציין **רשימה** – שבה כל פריט יכול להופיע כמה פעמים, ולכל פריט יש מיקום (אינדקס).
סדר הפריטים הוא סדר ההכנסה, אבל אפשר גם לבצע פעולות על פריטים באמצע הרשימה לפי אינדקס
מסויים – להוסיף (add), לשנות (set), לקרוא (get) ולהסיר (remove) ולמצוא את האינדקס של
פריט מסויים (indexOf). המימושים העיקריים של הממשק List הם ArrayList ו-
LinkedList. ההבדל ביניהם הוא בזמן הביצוע של פעולות שונות: ArrayList מאפשר לגשת
לאינדקס מסויים במהירות, אבל הסרה או הוספה עלולים לקחת הרבה זמן; ב-LinkedList לוקח זמן
להגיע לאינדקס מסויים, אבל אחרי שמגיעים לשם, הסרה או הוספה הם מהירים.

יש הרבה אלגוריתמים שאפשר לבצע על רשימות. אפשר לסדר רשימות (sort) לפי סדר כלשהו הנתון
ע"י Comparator. למשל, אם המשתנה c הוגדר ע"י List<String> c, אז אפשר לסדר את
המחרוזות בסדר עולה של אורך (מהקצרה לארוכה) ע"י:

```
c.sort((x,y)->x.length()-y.length());
```

אפשר גם לבצע פעולה כלשהי על כל פריט ברשימה (replaceAll). למשל, אפשר להכפיל כל מחרוזת
ברשימה ע"י:

```
c.replaceAll(x -> x+x);
```

בעזרת מתודות סטטיות של Collections, אפשר לבצע חיפוש בינארי על רשימה בהנחה שהיא כבר מסודרת (binarySearch), לחפש תת-רשימה בתוך רשימה נתונה (indexOfSubList), למלא רשימה בעותקים של פריט מסויים (fill), להעתיק רשימה לרשימה (copy), לסובב רשימה (rotate), להפוך רשימה (reverse), ולערבב רשימה בסדר אקראי (shuffle) – חשוב במיוחד לתיכנות משחקי קלפים...

תורים ומחסניות

הממשק Queue מציין **תור** – אפשר להכניס פריטים לסוף התור ע"י add ולהוציא פריטים מראש התור ע"י remove, הם ייצאו באותו סדר שבו נכנסו.

הממשק Deque מרחיב אותו ל**תור דו כיווני** – אפשר להכניס פריטים גם בראש התור ע"י push. כך אפשר להשתמש בממשק זה כמחסנית – להכניס ע"י push ולהוציא ע"י pop (שהוא זהה ל-remove). המימוש העיקרי של ממשק זה הוא ArrayDeque.

מימוש נוסף לממשק Queue הוא PriorityQueue המציין **תור עדיפויות**: מכניסים פריטים לתור ע"י add ומוציאים אותם ע"י remove, והם יצאו לפי הסדר הטבעי שלהם – הקטן ביותר ייצא ראשון. כמו ב TreeSet, גם כאן אנחנו יכולים לספק סדר שונה ע"י Comparator.

שימו לב: הפריטים בתור-עדיפויות לא נשמרים בצורה מסודרת – הם רק יוצאים לפי הסדר. אם חשוב לכם שהפריטים יישמרו בצורה מסודרת, השתמשו ב TreeSet. אם אתם רוצים גם לאפשר ערכים כפולים באותו אוסף – אתם צריכים MultiSet. כרגע אין דבר כזה בשפת ג'אבה; תצטרכו לממש לבד או לחפש חבילת קוד פתוח שיש בה MultiSet.

מפה

הממשק Map מציין **מפה**, שהיא התאמה בין מפתחות לערכים. בניגוד לשאר האוספים, למפה יש שני פרמטרי סוג – סוג המפתח וסוג הערך. למשל, אפשר להגדיר משתנה הממפה מחרוזות למספרים (שימוש אפשרי: ספר טלפונים):

```
Map<String,Integer> phoneBook = new HashMap<>();
```

במפה, כל מפתח יכול להופיע פעם אחת בלבד. המימושים השונים של מפה נבדלים בסדר שבו מופיעים המפתחות במפה, והם מקבילים למימושים השונים של קבוצה: במימוש HashMap הסדר הוא שרירותי, במימוש LinkedHashMap הסדר הוא אותו הסדר שבו הוכנסו הפריטים לקבוצה, ובמימוש TreeMap הסדר הוא הסדר ה"טבעי" של הפריטים – הסדר שנקבע ע"י המתודה compareTo של הפריטים או ע"י עצם מסוג Comparator המועבר לבנאי.

למפה נכנסים רק בזוגות של מפתח+ערך. כדי להכניס זוג משתמשים ב-put; אם קוראים לה פעמיים עם אותו מפתח, רק הערך השני יתעדכן. אפשר גם להשתמש ב-putIfAbsent; אם קוראים לה פעמיים עם אותו מפתח, הערך הראשון יישאר. כדי לקרוא את הערך המתאים למפתח מסויים משתמשים ב-get; אם הערך לא קיים יוחזר null. אפשר גם להשתמש ב getOrDefault ולהעביר ערך ברירת-מחדל שונה מ-null. לדוגמה:

```
phoneBook.put("Erel", 12);
phoneBook.put("Galya", 34);
phoneBook.put("Erel", 56);
phoneBook.putIfAbsent("Galya", 78);
System.out.println(phoneBook.get("Erel")); // 56
```

```
System.out.println(phoneBook.get("Galya")); // 34
System.out.println(phoneBook.get("xyz")); // null
System.out.println(phoneBook.getOrDefault("xyz",99)); // 99
```

למפה יש מתודות יותר מתוחכמות, המאפשרות לבצע חישובים כלשהם על ערכים. למשל, המתודה merge מקבלת מפתח, ערך, ופונקציה בין שני ערכים. היא משתמשת בפונקציה כדי למזג את הערך החדש עם הערך הקיים. שימוש אפשרי ל-merge הוא עבור מפה שסופרת. לדוגמה, נניח שאנחנו רוצים לספור כמה פעמים מופיעה כל מילה בקובץ מסויים. אנחנו יוצרים מפה בין מחרוזות למספרים:

```
Map<String,Integer> counter = new HashMap<>();
```

עכשיו, כשרואים מחרוזת חדשה, אנחנו רוצים להוסיף אותה למונה עם מספר 1, אבל אם המחרוזת כבר קיימת אנחנו רוצים להוסיף 1 לערך הקיים. פתרון אפשרי הוא לבצע לכל מילה:

```
counter.merge(word, 1, (x,y)→x+y )
```

איך עוברים על כל הפריטים במפה? יש כמה דרכים:

(א) אם רוצים לעבור רק על המפתחות, אפשר להשתמש במתודה keySet:

```
for (String k: phoneBook.keySet())
     System.out.println(k+" === "+phoneBook.get(k));
```

(ב) אם רוצים לעבור רק על הערכים, אפשר להשתמש במתודה values;

(ג) אם רוצים לעבור על המפתחות והערכים בו זמנית, אפשר להשתמש במתודה entrySet:

```
for (Map.Entry<String,Integer> e: phoneBook.entrySet())
     System.out.println(e.getKey()+" === "+e.getValue());
```

(ד) הדרך הכי קצרה (ויפה, לדעתי) היא להשתמש ב forEach :

```
phoneBook.forEach( (k,v) -> System.out.println(k+" === "+v) );
```

מבטים

אחד הדברים היפים בממשק Collection הוא שאפשר לממש אותו בצורה "קלילה" בלי לשמור את כל הפריטים בזיכרון. אוסף כזה נקרא גם **מבט** (view). למשל, אפשר ליצור מבט המייצג את כל המספרים השלמים בין שני מספרים נתונים:

```
public class Range implements Collection<Integer> {
     private int from;
     private int to;
     public Range(int from, int to) {
          this.from = from;
          this.to = to;
     }
}
```

אין צורך לשמור את כל המספרים בזיכרון – מספיק לשמור את שני הקצוות ולכתוב מימוש לכל המתודות של הממשק. אם אתם עובדים באקליפס ותעתיקו לשם את הקוד למעלה, תראו שאקליפס מסמן לכם קו אדום המציין שגיאת קומפילציה – השגיאה היא שלא מימשתם את כל המתודות של הממשק Collection. אקליפס מאפשר לכם להוסיף את כל המתודות הללו בלחיצת כפתור; לכם נשאר רק לכתוב את המימוש הנכון עבורן.

תרגיל כיתה: כיתבו מימוש לכל המתודות שאינן משנות את האוסף (size, contains, iterator...). השאירו את המתודות המשנות את האוסף (add, remove, clear...) ללא מימוש.

עוד דוגמאות למבטים: לממשק List יש מתודה subList המאפשרת לקבל מבט על תת-רשימה בין שני אינדקסים; לממשק SortedSet יש מתודה subSet המאפשרת לקבל מבט על תת-קבוצה-מסודרת בין שני פריטים (לפי הסדר הטבעי).

למחלקה Collections יש מתודות סטטיות המחזירות אוספים ריקים (למשל emptyList, emptySet), אוסף עם פריט יחיד (singleton, singletonList), nCopies המחזירה אוסף עם n עותקים של פריט נתון (הפריט כמובן נשמר רק פעם אחת).

חידה: כתבו מתודה המקבלת מספר n ומחזירה מחרוזת ובה n כוכביות. רמז: השתמשו ב- nCopies וב- String.join. קיראו בתיעוד כדי להבין מה כל מתודה עושה.

כל האוספים האלו בלתי ניתנים לשינוי - אם מנסים לשנות אותם ע"י add, remove וכד' נזרקת חריגה – UnsupportedOperationException.

אם יש לכם אוסף רגיל ואתם רוצים שאנשים לא יוכלו לשנות אותו, אתם יכולים ליצור לו מבט בלתי-ניתן-לשינוי, למשל:

```
Collections.unmodifiableList(list)
```

מחזירה מבט על list שאינו ניתן לשינוי: מי שמקבל אותו יוכל לקרוא את כל הפריטים שיש ברשימה המקורית, אבל אם ינסה לשנות, יקבל חריגה. זה מאד שימושי כשאתם רוצים להחזיר ערך ממתודה, אבל להבטיח שמי שיקבל אותו לא יוכל לשנות את המבנה הפנימי של העצם שלכם.

מקורות

- Cay Horstmann, “Core Java for the Impatient”, chapter 7.

סיכם: אראל סגל-הלוי.