

## פרק 8

# ממשקים

### א. היחס "מתפקד כ"

עד כה ראינו שתי דרכים לממש פולימורפיזם בג'אווה: ירושה כללית, או ירושה ממחלקות מופשטות. בשתי הדרכים הגדרנו מחלקת-על (מופשטת או מוחשית), המאגדת מאפיינים משותפים של תת-מחלקות אחדות. שימוש זה היה נוח כל עוד בין תת-המחלקות ובין מחלקת-העל אכן התקיים היחס "סוג של", המאפיין את מנגנון הירושה. למשל: מלבן, מעגל ומשולש – כל אחד מהם הוא סוג של צורה, ולכן היה טבעי להגדיר את המחלקות המייצגות אותם כתת-מחלקות של המחלקה "צורה". אולם, קיים גם יחס נוסף היכול להתקיים בין מחלקות שלהן התנהגות משותפת.

נסתכל על המחלקות הבאות: "מחרוזת", "מספר טבעי" ו"תאריך". ספק אם אפשר להגדיר לשלושתן מחלקת-על משותפת (למעט המחלקה Object כמובן). אף-על-פי-כן, נגלה כי יש להן יסוד משותף חשוב. שלושת סוגי העצמים הללו ניתנים למיון ולסידור: מחרוזות ניתן לסדר לפי סדר לקסיקוגרפי, כלומר לפי סדר האלפבית; מספרים טבעיים ניתן לסדר לפי גודלם; ותאריכים – לפי סדר כרונולוגי, כלומר מן המוקדם למאוחר. ככותבי מחלקות, נרצה להגדיר לכל מחלקה שיטת מיון.

האם באמת עלינו לכתוב שלוש שיטות שונות? עיון מעמיק יבהיר לנו שיש דרך אחרת. נניח כי כתבנו שיטה הממיינת מערך של מספרים טבעיים. כאשר נרצה לכתוב שיטה הממיינת מחרוזות, נוכל לבנות אותה באופן מקביל. ההבדל היחיד הוא אופן ההשוואה בין שני איברים בסיסיים: האם המספר 3 גדול מהמספר 5 או קטן ממנו? האם השם "כהן" מופיע בסדר המילוני לפני השם "לוי" או אחריו? במילים אחרות, השיטה צריכה לדעת רק מהו אופן ההשוואה בין שני עצמים מושווים.

רעיון זה סולל את הדרך להבנת היסוד המשותף בין העצמים: כל אחת מן המחלקות "מחרוזת", "מספר טבעי" ו"תאריך" מייצגת עצמים **ברי-השוואה**. יש משמעות והגיון בסדר ההופעה של כל סדרת עצמים ממחלקה כלשהי. השאלה מי מופיע לפני מי משמעותית עבור עצמים אלו אך היא חסרת משמעות עבור עצמים מטיפוס "צבע" למשל (אין משמעות לשאלה האם אדום מופיע לפני ירוק?).

לפיכך, נוכל להגדיר את היחס הבא בין המושגים "ניתן להשוואה" ו"תאריך": כל תאריך **יכול לתפקד כעצם ברי-השוואה**. אכן ניתן להשוות תאריך אחד למשנהו, אך זהו רק פן אחד של התאריך, והוא מעניין לצרכים מסוימים (למשל לצורך מיון). כאשר עסקנו בירושה, ראינו כי בין תת-מחלקה למחלקת-העל התקיים באופן טבעי היחס "סוג של" (למשל "כלב הוא סוג של חיה"). במקרה שלנו, לעומת זאת, מתקיים יחס חלש יותר. לא יהיה זה מדויק להגיד ש"תאריך" הוא סוג של עצם ברי-השוואה, שכן ההתנהגות "ברי-השוואה" מייצגת רק פן אחד של עצם התאריך, ולו פנים רבות אחרות, כגון: הוא מייצג זמן, ניתן לשכפול, ניתן להדפסה וכולי.

נסכם זאת כך: נאמר ש-A מתפקד כ-B, אם B מייצג התנהגות ש-A מקיים אותה, אך זאת אינה בהכרח ההתנהגות העיקרית או היחידה של A. בדרך כלל נתעניין ביחס זה רק כאשר B מייצג התנהגות המשותפת לכמה מחלקות (כגון הניתנות להשוואה). כמו כן, ראוי לציין כי השאלה האם A הוא סוג של B, או רק מתפקד כ-B, תלויה במידה רבה בתפקידים של A ו-B בתוכנית המסוימת שאנו כותבים.

בסעיפים הבאים נראה כיצד מנגנון הממשקים בג'אווה מאפשר לנו לייצג את היחס "מתפקד כ".

## ב. ממשקים בג'אווה

אחד המנגנונים המאפשר פולימורפיזם בג'אווה הוא הממשק (interface). בעוד שמנגנון הירושה אפשר לנו לייצג בתוכניותינו את היחס "סוג של", מנגנון הממשק מאפשר לנו, למעשה, לייצג את היחס "מתפקד כ".

הערה בטרם נתחיל: עד כה השתמשנו במילה "ממשק" כדי לציין חלק של תוכנית או מחלקה החשופים למשתמש. בפרק זה נשתמש במילה זו במובן אחר: לציון מנגנון נוסף בג'אווה, מנגנון ה-interface.

## הגדרת ממשק בג'אווה

ממשקים בג'אווה הם כלי המאפשר לנו להגדיר טיפוס המייצג התנהגות המשותפת לכמה מחלקות. בנוסף, ממשקים מאפשרים לנו להגדיר כמה טיפוסים התנהגות לאותה מחלקה. באופן מדויק, ממשק הוא אוסף חתימות של שיטות (ללא מימוש), המגדיר טיפוס התנהגות. ממשק יכול להכיל גם הצהרה על קבועים. מהגדרה זו תוכלו אולי להבין את הקשר בין המילה "ממשק" במובן הרגיל שלה, למילה "ממשק" כמציינת את המנגנון בג'אווה: הממשק בג'אווה מגדיר למעשה צורת התנהגות, מבלי לממש אותה.

בפרקים הקודמים ראינו כי מחלקה משמשת הן להגדרת טיפוס נתונים והן כתבנית שניתן ליצור ממנה עצמים או לבנות בעזרתה תת-מחלקות מפורטות יותר. כדאי לשים לב כי ממשק בג'אווה אינו מחלקה. הממשק אמנם מגדיר טיפוס נתונים, אך אי-אפשר ליצור ממנו עצמים, ומחלקות אינן יכולות לרשת ממנו.

נגדיר לדוגמה ממשק המייצג את ההתנהגות "להיות בר-השוואה". התנהגות זו כוללת שיטות השוואה: האם העצם הנוכחי "גדול" מעצם אחר (באחת המשמעויות של המילה "גדול")? האם הוא שווה לו? כאשר אנו מגדירים ממשק, איננו מעוניינים לממש את השיטות, אלא רק להגדיר אילו שיטות שייכות להתנהגות.

כדי להגדיר ממשק בג'אווה נשתמש במילה השמורה *interface*:

```
/**An interface defining objects that can be compared */
```

```
public interface Comparable{
```



```

/** returns true if and only if this object is equal to the given object */
public boolean isEqual (Object o);
/** returns true if and only if this object is less than the given object */
public boolean isLessThan (Object o);
}

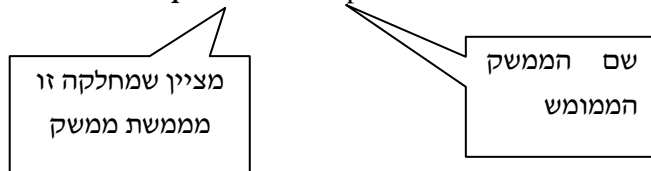
```

בדומה למחלקות, נשמור את הקוד שהממשק מוגדר בו בקובץ בעל שם זהה לשם הממשק. למשל, את הממשק Comparable, נשמור בקובץ ששמו Comparable.java. שימו לב, ב-Java API קיים ממשק בשם זהה (interface Comparable), והוא שונה במקצת מזה שהגדרנו כאן. לצורך הלימוד, נתייחס בפרק זה לממשק Comparable המוגדר כאן, ואילו בדפי העבודה, נתייחס לממשק Comparable כפי שהוא מופיע ב-Java API.

### מימוש ממשקים

לאחר שהגדרנו את הממשק, כל מחלקה יכולה לבחור האם **לממש** את הממשק. מימוש הממשק, פירושו שהמחלקה מצהירה שהיא אכן מבצעת את ההתנהגות שמגדיר הממשק. מימוש של ממשק נעשה על ידי שימוש במילה השמורה **implements**. למשל, כדי להגדיר שהמחלקה Date מממשת את הממשק Comparable, נכתוב את כותרת המחלקה באופן הבא:

```
public class Date implements Comparable
```



### מימוש ממשקים יוצר חוזה

בדומה לירושה ממחלקות מופשטות, מימוש ממשק דומה לחתימה על חוזה. כל מחלקה יכולה להגדיר כי היא מממשת ממשק מסוים (על ידי שימוש במילה **implements**), ובכך היא מצטרפת למעשה לחוזה. כמו בכל חוזה, למחלקה המממשת יש חובות וזכויות.

### חובות המחלקה המממשת

כל מחלקה המממשת את הממשק מתחייבת להגדיר שיטות בעלות חתימות זהות ל**כל** השיטות שהוגדרו בממשק. אם המחלקה המממשת לא תגדיר שיטות אלו, נקבל הודעת שגיאה בעת הידור קוד המחלקה.

לדוגמה, בקוד הבא של המחלקה Date יש מימוש לכל שיטות הממשק Comparable:

```

public class Date implements Comparable{
    private int day;
    private int month;
    private int year;
}

```

```

...
//other methods...
...
/** checks whether this date "is equal" to a given Object
 * the method assumes the given object is of type "Date"
 * implements the method in interface Comparable */
public boolean isEqual (Object o){
    Date d2 = (Date)o;
    boolean result = (this.day == d2.day) && (this.month == d2.month)
                    && (this.year == d2.year);

    return result;
}
/** ... */
public boolean isLessThan (Object o){
    Date d2 = (Date)o;
    boolean result = this.comesBefore(d2);
    return result;
}
}

```

מימוש כל השיטות  
שהוגדרו בממשק

---

? מדוע ביצענו בשיטות isEqual (Object o) ו-isLessThan(Object o) המרה כלפי מטה של הערך המתקבל?

---

### זכויות המחלקה המממשת

ראינו שכל מחלקה המממשת ממשק מתחייבת לממש את כל השיטות שהוגדרו בו. אולם מה מקבלת מחלקה מכך שהיא מממשת ממשק מסוים? כבר ציינו כי ניתן להגדיר בממשק קבועים במידה והוגדרו קבועים בממשק, המחלקה המממשת מקבלת אוטומטית את כל הקבועים שהוגדרו. אם למשל, הגדרנו בממשק את הקבוע `static final int MAX_PRICE = 500`, אזי קבוע זה יהפוך להיות תכונה (קבועה) של המחלקה המממשת. הנקודה החשובה בענין זה היא שמימוש הממשק מאפשר להתייחס אל עצמים מהמחלקה המממשת באמצעות טיפוס הממשק. למה הכוונה? כאמור, כל ממשק מגדיר טיפוס. כפי שציינו, ממשק אינו מחלקה ולא ניתן ליצור ממנו עצמים. אולם ניתן להמיר כל עצם מטיפוס מחלקה מממשת לטיפוס הממשק.

נניח למשל שהמחלקות Integer ו-Date מממשות את הממשק Comparable, ואילו המחלקה Turtle אינה מממשת ממשק זה. בשורות הקוד הבאות נבחן כיצד עובדה זו משפיעה על ההמרות שנבצע:

```
Comparable comp = new Comparable();
```


לא חוקי! לא ניתן ליצור  
עצמים מממשק

```
Date myBirthDay = new Date (30, 9, 1976);
```

חוקי: המחלקה Date מממשת את הממשק, ולכן ניתן להמיר עצמים מטיפוס זה לטיפוס הממשק

```
Comparable comp1 = myBirthDay;
```


```
Turtle t1 = new Turtle();
```

```
 Comparable comp2 = t1;
```

**לא חוקי!** המחלקה "צב" אינה מממשת את הממשק "ניתן להשוואה", ולכן לא ניתן להמיר עצמים ממנה לטיפוס זה

```
Date d1 = (Date)comp1;
```

חוקי: העצם comp1 נוצר במקור כעצם מטיפוס תאריך, ולכן זוהי המרה חוקית כלפי מטה

```
 Integer int1 = (Integer)comp1;
```


**לא חוקי!** המחלקה Integer אמנם מממשת את הממשק, אך comp1 הוגדר במקור כתאריך ולא ניתן להמיר אותו כלפי מטה למספר טבעי

## החזרה המוגדר מבטיח את תפקודו הנכון של טיפוס הממשק

כל עצם מטיפוס Comparable מכיר את כל שיטות הממשק ורק אותו. נסתכל למשל על הפקודות הבאות:

```
Date myBirthDay = new Date (30, 9, 1976);
```

```
Comparable comp1 = myBirthDay;
```

```
 int day = comp1.getDay();
```

```
boolean equals = comp1.isEqual (comp1);
```

לא חוקי! השיטה `getDay()` אינה שיטה של הממשק "ניתן להשוואה", ולכן לא ניתן להפעיל אותה בעזרת העצם `comp1`

חוקי: השיטה `isEqual(...)` היא שיטה של הממשק, ולכן ניתן להפעיל אותה בעזרת העצם `comp1`. השיטה שתזומן היא זו הממומשת במחלקה `Date`

כאשר אנו מקבלים עצם מטיפוס Comparable, אנו יודעים שהמחלקה שעצם זה נוצר לפיה אכן מממשת את הממשק של Comparable, כלומר מממשת את כל השיטות המוגדרות בו. כך מובטח לנו כי לכל קריאה לשיטה השייכת לממשק (כגון: `isEqual(...)`) יש כיסוי, והיא אחת משיטות העצם המדובר.

## מחלקה יכולה לממש מספר ממשקים

כאשר מחלקה A מממשת את הממשק I, עלינו לייצג את העובדה ש-A יכול לתפקד בתור I. I מייצג במקרה זה רק פן אחד או תפקיד אחד של A מני רבים קיימים. עיקרון זה אפשר ליישם בגיאווה משום שהמחלקה יכולה לממש כמה ממשקים יחד. למשל, המחלקה Integer יכולה לממש את הממשק "ניתן להשוואה" אך גם את הממשק "ניתן לחיבור". כדי לממש יותר ממשק אחד נכתוב את המילה *implements*, ואחריה את רשימת הממשקים שאנו רוצים לממש, מופרדים בפסיקים:

```
class A implements I
```

```
class A implements I, J, K
```

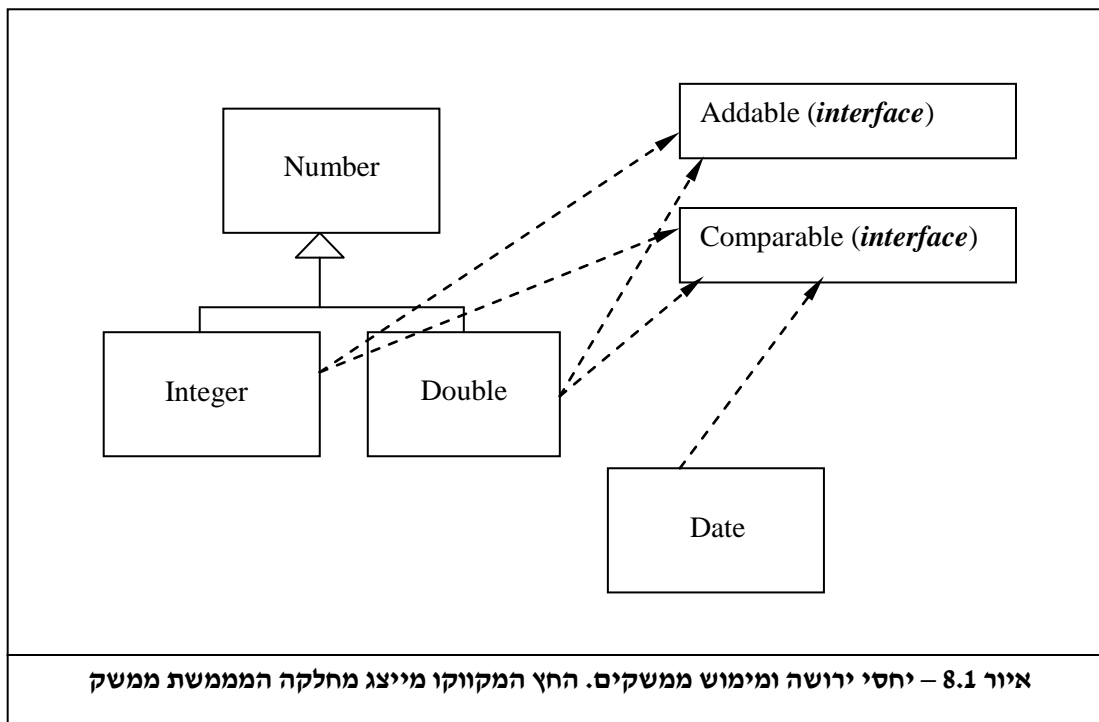
שימו לב שמחלקה יכולה לרשת ממחלקה אחרת וגם לממש ממשק (אחד או יותר):

```
class A extends B
```

```
class A extends B implements I
```

```
class A extends B implements I, J, K
```

```
class Integer extends Number implements Comparable, Addable
```



### ממשק יכול לרשת מממשק אחר

כדאי לדעת: ממשק יכול לרשת מממשק אחר. כאשר ממשק יורש מממשק אחר, פירוש הדבר שהממשק מקבל את כל רשימת החתימות של השיטות המופיעות בממשק-העל, וכן את הקבועים המוגדרים בו. כמו כן, אם הממשק J יורש מממשק I, נוכל להמיר כל משתנה מהטיפוס J למשתנה מהטיפוס I.

על אף הדמיון, אין לבלבל ירושה של ממשקים עם זו של מחלקות: ממשק לעולם אינו יכול לרשת ממחלקה, ומחלקה אינה יכולה לרשת מממשק. כדי לבצע ירושה של ממשקים נשתמש במילה *extends*, כפי שעשינו בירושה ממחלקות:

```
public interface I{
```

```
...
```

```
}
```

```
public interface J extends I{
```

```
...
```

```
}
```

### ג. ממשקים בשירות הפולימורפיזם

נחזור לדוגמה שהעסיקה אותנו בראשית הפרק. אנו רוצים לבנות שיטה כללית המקבלת מערך של עצמים כלשהם ומבצעת מיון של המערך לפי יחס השוואה כלשהו. נסתכל תחילה על השיטה

`sort (int[] numbers)` שיטה זו מקבלת מערך של מספרים שלמים, וממיינת אותו על פי אלגוריתם המיון המכונה "מיון בחירה" (selection sort):

```
public static void sort (int[] numbers){
    int minIndex;
    //while the unsorted part is not last place in list:
    for (int placeIndex = 0; placeIndex < numbers.length-1; placeIndex++){
        minIndex = findMinIndex (numbers, placeIndex);
        swap (numbers, minIndex, placeIndex);
    }
}
/*
 * Swaps between the number that exists in the index1 index in the array, and the number that
 * exists in the index2 index in the array.
 */
private static void swap (int[] numbers, int index1, int index2){
    int temp = numbers[index1];
    numbers[index1] = numbers[index2];
    numbers[index2] = temp;
}
/*
 * Returns the index of the minimal number in the array.
 */
private static int findMinIndex(int[] numbers, int placeIndex){
    int minNum = numbers[placeIndex];
    int minIndex = placeIndex;
    for (int i = placeIndex + 1; i < numbers.length; i++){
        if (numbers[i] < minNum){
            minNum = numbers[i];
            minIndex = i;
        }
    }
    return minIndex;
}
```


שימו לב, השיטה `sort(...)` משתמשת בשתי שיטות עזר פרטיות, ולכן השיטה עצמה קצרה יותר, והקוד נוח לקריאה ולמעקב.

---

? כתבו שיטה מקבילה הממיינת מערך של עצמים מטיפוס `Date` (תוכלו להשתמש בממשק המחלקה `Date` כפי שהוגדר בפרק 4). מה שונה ומה דומה בין השיטה שכתבתם והשיטה הממיינת מספרים טבעיים?

---

כאמור, לא ניתן ליצור עצמים מממשק, כלומר אי-אפשר להשתמש במלה השמורה `new` אם אחריה מופיע שם של ממשק:

 `Comparable comp = new Comparable();`

אין להסיק מכך שאי-אפשר להתייחס לעצמים מטיפוס הממשק. לדוגמה, שיטה יכולה לקבל פרמטר מטיפוס ממשק ולקרוא לשיטות שלו, בהנחה שהפרמטר שיועבר לשיטה יהיה מטיפוס הממשק את הממשק.



נניח שבמחלקת השירות Comparing אנו רוצים להגדיר שיטה סטטית, שתקבל שני עצמים כפרמטרים, ותדפיס את הגדול מביניהם. ברור ששני הפרמטרים צריכים להיות ברי-השוואה. לכן חתימת השיטה תהיה:

```
public static void printLarger (Comparable first, Comparable second)
```

במימוש השיטה עלינו להשוות בין שני הפרמטרים, למצוא את הגדול מביניהם ולהדפיסו:

```
public static void printLarger (Comparable first, Comparable second){  
    Comparable bigger = first;  
    if (first.isLessThan (second)) {  
        bigger = second;  
    }  
    System.out.println (bigger);  
}
```

בזימון השיטה printLarger(...) יש להעביר לה פרמטרים שהם מופעים של מחלקות המממשות את הממשק Comparable, לדוגמה:

```
Date myBirthDay = new Date (8, 12, 1974);  
Date yourBirthDay = new Date (30, 9, 1976);  
Comparing.printLarger (myBirthDay, yourBirthDay);
```

דוגמה נוספת היא יצירת מערך של עצמים המממשים כולם אותו ממשק. לדוגמה, השורה הבאה היא חוקית:

```
Comparable[] compArray = new Comparable[4];
```

במערך compArray יכולים להיות מופעים של מחלקות המממשות את הממשק Comparable. מסיבה זו, ניתן גם להעביר פרמטר כגון Comparable[] things לשיטה, כפי שמופיע להלן. נכליל את השיטה sort(...) בעבור עצמים "ניתנים להשוואה" (Comparable) כלשהם:

```
public static void sort (Comparable[] things){  
    int minIndex;  
    //while the unsorted part is not last place in list:  
    for (int placeIndex = 0; placeIndex < things.length-1; placeIndex++){  
        minIndex = findMinIndex (things, placeIndex);  
        swap (things, minIndex, placeIndex);  
    }  
}
```

```
private static void swap (Comparable[] things, int index1, int index2){  
    Comparable temp = things [index1];  
    things [index1] = things [index2];  
    things [index2] = temp;  
}
```

```
private static int findMinIndex (Comparable[] things, int placeIndex){
```

```

Comparable minThing = things [placeIndex];
int minIndex = placeIndex;
for (int i = placeIndex + 1; i < things.length; i++){
    if (things [i].isLessThan (minThing)){
        minThing = things [i];
        minIndex = i;
    }
}
return minIndex;
}

```

**?** כעת השוו שיטה זו לשיטה הממיינת את העצמים מטיפוס "תאריך" שכתבתם כתשובה לשאלה הקודמת. במה הן דומות ובמה הן שונות? מה היתרון של השיטה החדשה?

שיטה זו מנצלת את העובדה ששלחנו לה מערך כלשהו של עצמים ניתנים להשוואה. נוכל להפעיל את השיטה על המערך birthdays המכיל תאריכים של ימי הולדת של חברים, על המערך phoneNums המכיל מספרי טלפון או על המערך names המכיל שמות. בכל מקרה, שיטת ההשוואה תיקבע לפי המחלקה שממנה נוצרו העצמים במערך, ותמומש בהתאם. במקום לכתוב שלוש שיטות שונות: sortDates(...), sortIntegers(...) ו-sortStrings(...), כתבנו שיטת מיון אחת כללית.

אף שזהו פתרון כללי, ראוי לשים לב שבמימוש שיטות ההשוואה של רוב המחלקות, הנחנו כי הן משוות שני עצמים מהמחלקה עצמה (הנחה זו מתבטאת בביצוע המרה כלפי מטה, לעצם מהמחלקה). לכן, סביר ששיטת המיון תתפקד רק כאשר אנו מנסים להשוות עצמים מאותה המחלקה, ולא מערך מעורב של עצמים ממחלקות שונות. הגבלה זו היא הגיונית למדי: בדרך כלל לא ננסה להשוות תפוזים לפילים...

כמובן, נוכל להרחיב את השימוש בשיטה sort(...) בעבור עצמים רבים נוספים בעתיד. אם נרצה להגדיר מחלקה חדשה שאף היא תשתמש בשיטת המיון, יהיה עלינו רק להגדיר את המחלקה כך שתממש את הממשק "ניתן להשוואה", ושיטת המיון שהגדרנו (ללא כל שינוי!) תתאים גם לעצמים מן המחלקה החדשה.

#### ד. סיכום: מה בין ירושה למימוש

בשני הפרקים האחרונים הכרנו שני מנגנונים מעניינים של גיאווה למימוש פולימורפיזם: ממשקים ומחלקות מופשטות. שני מנגנונים אלו דומים מבחינות מסוימות: שניהם כוללים חלק מופשט שאינו ממומש. שניהם יוצרים חוזה, שאותו צריכות למלא המחלקות היורשות או המחלקות המממשות. ולבסוף, שניהם מגדירים טיפוס נתונים, שלא ניתן ליצור ממנו עצמים, אך ניתן להציב בו עצמים מתאימים.

למרות הדמיון, כדאי לשים לב להבדלים בין שני המנגנונים המסוכמים בטבלה הבאה:

המחלקה A <b>יורשת</b> מ-B	המחלקה A <b>מממשת</b> את I
B היא מחלקה (מוחשית או מופשטת)	I הוא ממשק

מייצג יחס "סוג של" (כלומר הוא סוג של צורה)	מייצג יחס "מתפקד כ" (כלומר מתפקד כ"ניתן לציור", "ניתן להשוואה" וכו')
ניתן לרשת רק ממחלקה אחת (בג'אווה אין ירושה מרובה)	מחלקה אחת יכולה לממש ממשקים רבים כרצוננו
גם אם B מחלקה מופשטת, ניתן להגדיר לה תכונות (לא קבועות), ולממש בה שיטות	לא ניתן להגדיר ב-I תכונות לא קבועות, ולא ניתן לממש בו שיטות. בממשק ניתן להגדיר קבועים ולהצהיר על שיטות
B מייצגת את הפן העיקרי של A	I מייצג פן אחד (אולי מני רבים) של A

### מושגים

<i>interface</i>	ממשק (בג'אווה)
functions as	היחס "מתפקד כ"
<i>implements</i>	(מחלקה) מממשת (ממשק)

## פרק 8 דף עבודה מס' 1

### תרגיל תיאורטי

#### מטרות

1. תרגול המושג ממשק.
2. תרגול ההבדל בין ממשק למחלקה.

#### מה עליכם לעשות?

לפניכם קטע קוד ללא שגיאות.

1. כתבו את A במלואו בהנחה כי הוא ממשק (*interface*).
2. כתבו את המחלקות B ו-C.

אין לממש את השיטות של המחלקות אלא רק לכתוב את החתימות שלהן.  
אם למחלקות יש תכונות – יש לכתוב גם אותן.

```
public static void main(String[] args){  
    int x;  
    String y;  
    C b1 = new B();  
    A[] arrA = new A[3];  
    arrA[0] = new B (x,y);  
    arrA[1] = new B();  
    arrA[2] = new C();  
    arrA[0].f();  
    arrA[2].g();  
    arrA[1].h();  
}
```

1. שרטטו תרשים בעזרת UML המציג את הקשרים המדויקים בין A, B ו-C. הסבירו על פי מה אתם מסיקים את הקשרים הללו. האם ייתכנו קשרים אחרים?
2. האם C יכול להיות ממשק? אם כן, כתבו אותו כממשק. אם לא, נמקו מדוע.

**בהצלחה!**

## פרק 8 דף עבודה מס' 2

### מטוסים ורכבות

#### מטרות

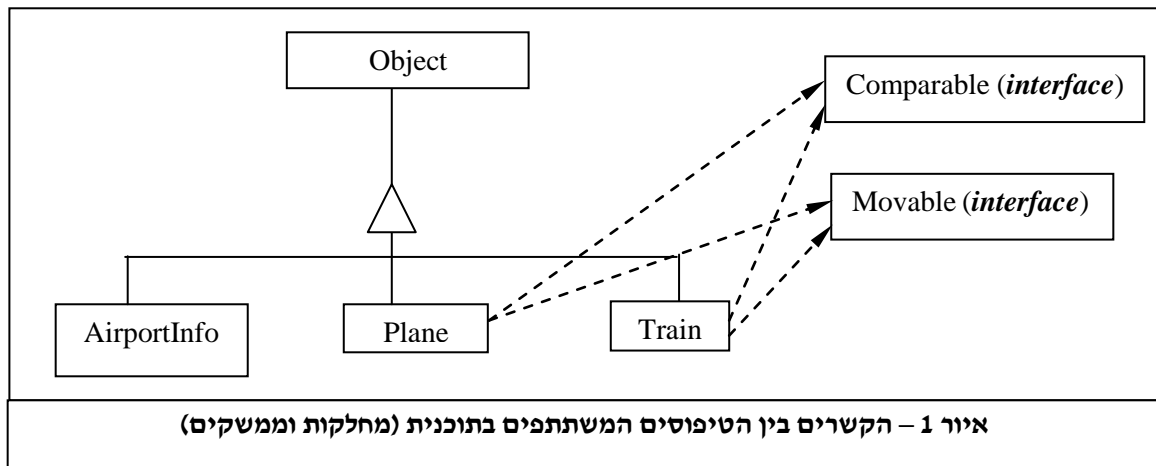
1. מימוש ממשק ומימוש מחלקה יורשת בתרגיל אחד.
2. מימוש הממשק Comparable מתוך ה-Java API.
3. תרגול השימוש בעקרון הפולימורפיזם על ידי שימוש בממשקים.

#### רקע

- כדי לדעת אילו שיטות עליכם לכתוב על מנת לממש את הממשק Comparable, עיינו בפירוט שלו המופיע ב-Java API.
- בתרגיל זה עליכם להשתמש בשיטה `sort (Object[] a)` `static void` של המחלקה Arrays. המחלקה Arrays כלולה ב-Java API. עליכם לציין על ידי שימוש בפקודה `import` שברצונכם להשתמש בשיטה זו. כדי ללמוד יותר על שיטה זו – עיינו בממשק המחלקה Arrays המופיע ב-Java API.

#### מה עליכם לעשות?

עליכם לכתוב תוכנית שתאפשר לדמות מערכת מידע של נמל תעופה. מערכת המידע תאפשר למשתמש לקבל מידע אודות תנועת המטוסים והרכבות בנמל. יחסי הטיפוסים המשתתפים בתוכנית מתוארים בתרשים הבא:



מותר לכם לשנות את התרשים המופיע באיור 1, ובלבד שיחסי הטיפוסים המתוארים בו יישמרו.

בסיום תוכלו לבדוק את התוכנית על ידי המחלקה `TestAirportInfo`, שכתבנו בעבורכם. מחלקה זו כוללת את השיטה `main(...)`, והיא משתמשת במחלקות שכבר כתבתם. את המחלקה תוכלו למצוא בדיסק המעבדות בתיקייה:

Chap8\Airport

## השוואה בין עצמים

השוואה בין העצמים נעשית על פי הקריטריונים הבאים :

- מטוסים – הגובה המרבי לטיסה.
- רכבות – מספר הנוסעים המרבי ברכבת.
- מסלולי המראה – אורך המסלולים.

## הממשק Movable

מגדיר טיפוס של אובייקט היכול לנוע ממקום למקום. פירוט הממשק :

חתימת השיטה	תיאור השיטה
String getSource()	מחזירה את נקודת המוצא של כלי התחבורה
String getDestination()	מחזירה את היעד של כלי התחבורה
void move()	מחליפה בין המוצא של כלי התחבורה ליעד שלו
String getType()	מחזירה את שם הסוג של אובייקט כלי התחבורה
String details()	מחזירה פרטים על אודות כלי התחבורה

## המחלקה Plane

מחלקה זו מגדירה אובייקט מטיפוס "מטוס", ומממשת את הממשקים Movable ו-Comparable. תכונות המטוס הן: מספר הרישוי של המטוס, נקודת המוצא של מסלולו, היעד, הגובה המרבי לטיסה.

## המחלקה Train

מחלקה זו מגדירה אובייקט מטיפוס רכבת, ומממשת את הממשקים Movable ו-Comparable. תכונות הרכבת הן: מספר הרישוי של הרכבת, נקודת המוצא של מסלולה, היעד, מספר הנוסעים המרבי ברכבת.

## המחלקה AirportInfo

זו מחלקת שירות הכוללת שיטות סטטיות המאפשרות למיין כלי תחבורה מסוגים שונים, ולהדפיס דוחות על תנועתם בנמל. עליכם לממש את הממשק הבא :

חתימת השיטה	תיאור השיטה
static void sortTransport (Comparable[] transport)	ממיינת מערך של אובייקטים מטיפוס Comparable לפי "מפתח ההשוואה" שהוגדר בעבורם
static String reportAll (Movable[] movable)	מחזירה את הדוחות של כל האובייקטים מטיפוס Movable, שהיא מקבלת (ראו פירוט בהמשך)

כדי לממש את השיטה `sortTransport(...)`, השתמשו בשיטה הסטטית `sort (Object[] a)` של המחלקה `Arrays` הנמצאת ב-Java API.

השיטה `reportAll(...)` תחזיר את המחרוזות הבאה הכוללת את הדוחות של כל האובייקטים:

Movable Type: <type>

Plane/Train number: <plane/train number> (בהתאם לסוג)

Passengers/Altitude: <passengers>/<altitude> (בהתאם לסוג)

Source: <source>

Destination: <destination>

<empty line>

Movable Type: <type>

Plane/Train number: <plane/train number> (בהתאם לסוג)

Passengers/Altitude: <passengers>/<altitude> (בהתאם לסוג)

Source: <source>

Destination: <destination>

...

## הרצת הפרויקט

כדי להריץ את הפרויקט ולבדוק את המחלקות שכתבתם צרפו אליו את המחלקה `TestAirportInfo`, שכתבנו בעבורכם. עיינו היטב בקוד המחלקה. המחלקה כוללת את השיטה `main(...)`, והיא תאפשר לכם לבדוק את המחלקות שכתבתם. לאחר מכן הדרו את הפרויקט הכולל את המחלקה והריצו את התוכנית. את המחלקה תוכלו למצוא בדיסק המעבדות בתיקיה:

Chap8\Airport

## שאלות

ענו בקצרה על כל אחת מהשאלות הבאות:

1. באילו ממשקים השתמשתם בתרגיל זה?
2. איזה עיקרון של תכנות מונחה אובייקטים יכולתם להפעיל משום שמימשתם את הממשקים?
3. האם הייתם יכולים להשיג אותה תוצאה על ידי שימוש במנגנון ירושה או במחלקות מופשטות? אם כן, הסבירו כיצד. אם לא, הסבירו מדוע.
4. מה היה קורה אילו במחלקה `Plane` הייתם מממשים רק את השיטה `details()` מתוך הממשק `Movable`? האם הייתם מצליחים להדר את התוכנית? הסבירו.

בשאלות הבאות סמנו את התשובה הנכונה (רק תשובה אחת נכונה):

5. מהו ממשק (*interface*)?

- א. טיפוס המהווה אוסף של הצהרות על קבועים ושיטות.
  - ב. אוסף של שיטות פומביות של מחלקה.
  - ג. מחלקה שתת-מחלקה יכולה לפרט (*extends*).
  - ד. מנגנון הקובע היררכיה בין מחלקות.
6. האם ממשק יכול להכיל מימוש של שיטה?

- א. לא.
- ב. לפעמים.
- ג. כן.
- ד. תמיד.

7. האם ניתן להשתמש בשם של ממשק (*interface*) כדי להגדיר טיפוס משתנה. לדוגמה כך:

```
public static void main (String[] args){  
    SomeInterface x;  
    ...  
}
```

- א. לא, משתנה חייב להיות מטיפוס בסיסי כמו *int*, *double* או *boolean*.
- ב. לא, משתנה חייב להיות מוגדר כטיפוס בסיסי או כהפניה לטיפוס של אובייקט.
- ג. כן, המשתנה *x* יכול להיות מוגדר כהפניה לכל אובייקט שהמחלקה שלו מממשת את הממשק.
- ד. לא, משתנה חייב להיות מוגדר כהפניה לטיפוס של אובייקט.

**בהצלחה!**