ממשקים

ממשק הוא אוסף של כותרות של מתודות, בלי גוף.

בשיעור הקודם למדנו על מחלקה – למחלקה יש מתודות עם גוף שמגדיר איך הן פועלות, ויש גם שדות (משתנים). אבל למה צריך ממשק, שיש בו רק מתודות, ובלי גוף? זה לא נשמע שימושי במיוחד!

למעשה, ממשק הוא דבר מאד שימושי. הוא מאפשר לנו לכתוב **אלגוריתם** כללי, שיכול לעבוד על הרבה מחלקות שונות.

למשל, נניח שאנחנו רוצים לכתוב מתודה לציור גרף של הפונקציה y=sin(x), בתחום בין 0 ל-10. הרעיון הוא לקחת 100 ערכי x שונים בתחום, לכל ערך x לחשב את ערך y המתאים, ואז לשים נקודות בכל ה-(x,y) שחישבנו. כיוון ששירטוט גרפים בג'אבה זה סיפור מסובך, נציג קודם-כל קוד שמחשב את הנקודות וכותב אותן למסך:

```
void printPoints(double xFrom, double xTo, int numPoints) {
     double difference = (xTo-xFrom)/numPoints;
     for (double x = xFrom; x<=xTo; x+=difference) {
          double y = Math.sin(x);
          System.out.println("("+x+","+y+")");
     }
}
```

(בגיטהאב, בתיקייה lesson2, יש קוד דומה שאמור גם לצייר את הגרף. הקוד נבדק על המחשבים בכיתות הלימוד, שיש בהם Java 8 ו-Eclipse Neon על חלונות 10. הקוד נבדק גם באובונטו עם Java 9 ו-Eclipse Oxygen. במערכות אחרות ייתכן שצריך להתקין תוספים שונים).

עכשיו, אנחנו רוצים לצייר גרף של פונקציה אחרת – y=tan(2x). הקוד הוא כמעט זהה – השורה היחידה שצריך לשנות היא השורה שבה מחשבים ערך של y המתאים לערך מסויים של x. במקום לכתוב:

```
y = Math.sin(x);
```

צריך לכתוב:

```
y = Math.tan(2*x);
```

חבל לכתוב את אותו קוד פעמיים רק בשביל לצייר פונקציה אחרת, נכון?

בדיוק בשביל זה המציאו את ה**ממשק**. אנחנו ניצור ממשק חדש עם מתודה אחת בלבד בלי גוף:

```
public interface Function {
     double apply(double x);
}
```

עכשיו, נשנה את המתודה לציור גרף של פונקציה, כך שתקבל פרמטר נוסף f מסוג Function:

```
void printPoints(Function f, double xFrom, double xTo, int
numPoints) { … }
```

ונשנה את השורה המחשבת את ערך y ל:

```
y = f.apply(x);
```

כל עצם מסוג Function, שיגיד לנו איך לממש את המתודה apply – נוכל לצייר אותו!

איך מגדירים עצם מסוג Function? אם ננסה פשוט לאתחל עצם כזה ע"י new:

```
Function f1 = new Function();
```

זה לא יעבוד – נקבל הודעת שגיאה. אנחנו מנסים להגדיר עצם מסוג Function ולא אומרים איך הוא מממש את הפונקציה apply!

נראה כאן שלוש דרכים להגדיר עצם שמממש ממשק.

דרך א: מחלקה עם שם

נגדיר מחלקה חדשה (בקובץ חדש), ונצהיר שהיא מממשת את הממשק. מילת המפתח היא implements. הנה דוגמה:

```
public class Sine implements Function {
     public double apply(double x) {
          return Math.sin(x);
     }
}
```

עכשיו אפשר לכתוב בתוכנית הראשית:

```
printPoints(new Sine(), 0, 10, 100)
```

ונקבל נקודות המתאימות לפונקציית סינוס.

במחלקה החדשה יכולים להיות גם שדות נוספים שאינם קשורים לממשק, למשל, המחלקה הבאה מממשת את הפונקציה sin(ax) כאשר a הוא קבוע המועבר לבנאי:

```
public class SineAX implements Function {
     private double a;
     public SineAX(double a) { this.a = a; }
     public double apply(double x) {
          return Math.sin(a*x);
     }
}
```

עכשיו אפשר לכתוב בתוכנית הראשית:

```
printPoints(new Sine(5), 0, 10, 100)
```

ונקבל נקודות המתאימות לפונקציה sin(5x).

תרגיל כיתה: קחו את המחלקה Monom שהגדרנו בשיעור הקודם, ועשו בה את השינויים הדרושים כך שיהיה אפשר להשתמש בה כארגומנט למתודה printPoints.

דרך ב: מחלקה בלי שם

אפשר להגדיר מחלקה חדשה בתוכנית הראשית, בלי לתת לה שם (מחלקה כזאת נקראת "מחלקה אנונימית", anonymous class). זה נראה כך:

```
Function sine = new Function() {
     public double apply(double x) {
          return Math.sin(x);
     }
};
printPoints(sine, 0, 10, 100)
```

דרך ג: ביטוי-חץ ("למדא")

אפשר להגדיר את אותה מחלקה שהגדרנו למעלה בקיצור, בשורה אחת, ע"י ביטוי-חץ:

```
Function sine =  x -> Math.sin(x);
```

ביטוי-חץ נקרא בלועזית "ביטוי למדא", lambda expression. ניתן להשתמש בו בכל מקום שבו רוצים לממש ממשק עם מתודה אחת בלבד (כמו הממשק Function).
כשהקומפיילר רואה ביטוי חץ, הוא עושה הרבה דברים מאחרי הקלעים: הוא מגדיר מחלקה אנונימית חדשה הממשת את הממשק המתאים (Function), יוצר בתוכה מתודה אחת בשם המתאים (apply), המתודה מקבלת ארגומנט אחד בשם x, ומחזירה את Math.sin(x).

ביטוי-חץ הוא דרך מאד נוחה להעביר פעולות למתודות. לדוגמה, בעזרת ביטוי-חץ אפשר לצייר פונקציות רבות באותה תוכנית:

```
printPoints( x->Math.sin(x), 0, 10, 100);
printPoints( x->x*x, 0, 10, 100);
printPoints( x->1/Math.tan(x), 0, 10, 100);
printPoints( x->x*Math.log(x), 0, 10, 100);
```

ועוד. כתבנו את המתודה printPoints פעם אחת, ואנחנו משתמשים בה הרבה פעמים עם פונקציות שונות.

הערה: אם בממשק יש שתי מתודות או יותר, אי-אפשר להשתמש בביטוי-חץ . צריך להשתמש בדרך א או בדרך ב.

עוד דברים שאפשר לעשות בעזרת ממשקים

תרגיל א. כתבו מתודה repeat שמקבלת פעולה כלשהי ומספר שלם חיובי n, ומבצעת את הפעולה n פעמים.
פתרון: נגדיר ממשק עם מתודה אחת המבצעת פעולה. הנה ממשק כזה:

```
public interface Runnable {
     void run();
}
```

(למעשה אין צורך להגדיר את הממשק הזה, הוא כבר מוגדר בג'אבה).
עכשיו אפשר לכתוב את המתודה:

```
static void repeat(int n, Runnable action) {
     for (int i=0; i<n; i++)
          action.run();
}
```

ולהשתמש בה למשל כך (עם "ביטוי חץ"):

```
repeat(10, () ->  System.out.print("x"))
```

המשמעות של ביטוי-החץ היא: "הגדר מתודה בלי ארגומנטים (), שמדפיסה x".

תרגיל ב. כתבו מתודה filter המקבלת מערך של מחרוזות ותנאי כלשהו לסינון המערך, ומחזירה מערך חדש מסונן.

פתרון: נגדיר ממשק עם מתודה אחת הבודקת תנאי כלשהו על מחרוזת:

```
public interface Predicate {
     boolean test(String s);
}
```

ואז נכתוב את המתודה לסינון:

```
static List<String> filter(List<String> strings, Predicate p) {
     List<String> output = new ArrayList<>();
     for (String s: strings)
          if (p.test(s))
               output.add(s);
     return output;
}
```

ונשתמש בה למשל כך:

```
List<String> strings = new ArrayList<>();
Collections.addAll(strings, "aaa","cc","bbbb","abc");
List<String> startWithA = filter(strings, s->s.startsWith("a") );
```

תרגיל ג. כתבו מתודה המקבלת אוסף מחרוזות ומסדרת את המחרוזות לפי תנאי כלשהו.
פתרון: נשתמש בממשק שכבר קיים בג'אבה – הממשק Comparator. בממשק זה יש מתודה אחת compare המקבלת שני ארגומנטים, x ו-y, ומחזירה מספר שלם המציין מי מהם צריך להיות ראשון בסדר: אם הערך המוחזר הוא שלילי אז x צריך להיות ראשון, אם הערך המוחזר הוא חיובי אז y צריך להיות ראשון, ואם הערך המוחזר הוא אפס אז הם שקולים (לא משנה מי ראשון). למשל, כך אפשר לסדר מחרוזות לפי האורך, מהקצרה לארוכה:

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

חידה: איך נסדר את המחרוזות מהארוכה לקצרה?

הרחבת ממשקים - ירושה

לפעמים אנחנו רוצים ליצור ממשק שהוא דומה לממשק שכבר קיים, אבל מוסיף לו עוד מתודות.

נחזור לדוגמה של הפונקציה: נניח שאנחנו רוצים לכתוב מתודה שמציירת פונקציה, ויחד איתה גם את הנגזרת שלה. המתודה הזאת עובדת רק על פונקציה שיש לה נגזרת – רק על פונקציה **גזירה**. כדי לממש מתודה כזאת אנחנו צריכים לכתוב ממשק המייצג פונקציה גזירה. כל פונקציה גזירה היא גם פונקציה – היא צריכה לדעת לחשב את הערך של עצמה בנקודה (apply). אבל פונקציה גזירה יודעת גם להחזיר את הנגזרת של עצמה. לכן הממשק של פונקציה גזירה **מרחיב** את הממשק של פונקציה. בג'אבה מילת המפתח היא extends:

```
public interface DifferentiableFunction extends Function {
     Function getDerivative();
}
```

הערה: בשפות אחרות אומרים שהממשק של פונקציה גזירה **יורש** (inherits) את הממשק של פונקציה (כי הוא מכיל גם את המתודה apply הנמצאת בממשק Function). בשפת ג'אבה, במקום יורש אומרים מרחיב – במקום inherits אומרים extends.

עכשיו נכתוב את המתודה שרצינו:

```
static void printFunctionAndDerivative(DifferentiableFunction f,
     double xFrom, double xTo, int numPoints) {
     printPoints(f, xFrom, xTo, numPoints);
     printPoints(f.getDerivative(), xFrom, xTo, numPoints);
}
```

שימו לב ש-f עוברת כארגומנט למתודה printPoints. המתודה printPoints מצפה לקבל ארגומנט מסוג Function, ו-f היא מסוג DifferentiableFunction, אבל הממשק הזה מרחיב את Function ולכן ניתן להשתמש כאן ב-f. זה גם הגיוני: כל פונקציה גזירה היא פונקציה, ולכן אפשר להעביר פונקציה גזירה לכל מתודה שמצפה לקבל פונקציה.

כדי לממש את הממשק המורחב DifferentiableFunction צריך לממש שתי מתודות, ולכן אי-אפשר לעשות זאת בעזרת ביטוי-חץ. אפשר לעשות זאת למשל ע"י הגדרת מחלקה חדשה (המחלקה משתמשת במחלקה Sine שהגדרנו קודם):

```
public class Cosine implements DifferentiableFunction {
     public double apply(double x) {
          return Math.cos(x);
     }
     public Function getDerivative() {
          return new Sine();
     }
}
```

מושג ההרחבה (=ירושה) קיים לא רק לגבי ממשקים אלא גם לגבי מחלקות. הרעיון דומה:

- כותבים מחלקה חדשה אשר מרחיבה (**extends**) מחלקה קיימת.
- המחלקה החדשה יורשת את כל המתודות והשדות של המחלקה הקיימת, ויכולה להוסיף עליהם עוד מתודות ושדות.
- ניתן להשתמש במחלקה החדשה כארגומנט לכל מתודה המצפה לקבל את המחלקה הקיימת.

יש הבדל אחד בין הרחבת ממשק להרחבת מחלקה: ממשק יכול להרחיב כמה ממשקים בו-זמנית, אבל מחלקה יכולה להרחיב רק מחלקה אחת.

עוד על ירושה של מחלקות ניתן לקרוא בפרק 4 בספר, וכן באחד השיעורים הבאים.

הכללת ממשקים – תכנות גנרי

אפשר לכתוב ממשקים שיעבדו עם הרבה סוגים של עצמים שונים בו-זמנית.

נחזור לדוגמה של הסינון: כתבנו למעלה מתודה בשם filter שמסננת רשימה של מחרוזות. אבל האלגוריתם של סינון הוא אותו אלגוריתם בין אם מסננים מחרוזות, מספרים או אובייקטים כלשהם. זה די פשוט להכליל את התנאי ואת המתודה שיטפלו בכל סוג שהוא. קודם-כל נכליל את הממשק ע"י הוספת פרמטר-סוג בסוגריים משולשים "<T>":

```
public interface Predicate<T> {
     boolean test(String s);
}
```

עכשיו נכליל את המתודה לסינון ע"י שימוש בפרמטר סוג דומה:

```
static <T> List<T> filter(List<T> items, Predicate<T> p) {
     List<T> output = new ArrayList<>();
     for (T s: items)
          if (p.test(s))
               output.add(s);
```

```
    return output;
}
```

אפשר להשתמש במתודה לסינון רשימת מחרוזות בדיוק כמו קודם, אבל עכשיו אפשר להשתמש באותה מתודה כדי לסנן רשימה של מספרים, למשל כך:

```
List<Integer> ints = new ArrayList<>();
Collections.addAll(ints, 1,9,2,8,3,7,4,6,5,5);
List<Integer> above5 = filter(strings, x -> x>=5 );
```

אפשר לסנן רשימות מכל סוג שהוא בעזרת אותה פונקציה.

ניתן להכליל לא רק ממשקים אלא גם מחלקות. הרעיון דומה:

- כותבים מחלקה חדשה עם פרמטר-סוג בסוגריים משולשים.
- כשיוצרים עצם מהמחלקה החדשה, צריך לשים סוג ספציפי בתוך הסוגריים המשולשים.

עוד על תיכנות גנרי ופרמטרי-סוג ניתן לקרוא בפרק 6 בספר.

ממשקים קיימים בג'אבה

בשפת ג'אבה כבר מוגדרים הרבה ממשקים פונקציונליים (- עם מתודה אחת), וניתן להשתמש בהם בתוכניות שלכם. לפרטים חפשו בגוגל "Java util function". כמעט כל הממשקים הם גנריים ומקבלים פרמטר-סוג אחד או יותר.

חידה: אנחנו רוצים לשנות את המתודה repeat שכתבנו למעלה, כך שתקבל פעולה עם פרמטר ותעביר לפעולה את האינדקס הרץ (i). למשל, הקריאה:

```
repeat(10, i ->  System.out.print(i))
```

תדפיס את כל המספרים בין 0 ל-9. באיזה ממשק מהממשקים הקיימים בג'אבה כדאי להשתמש? ואיך נשנה את המתודה repeat?

מקורות

- Cay Horstmann, "Core Java for the Impatient", chapter 3 (interfaces), chapter 4 (inheritance), chapter 6 (generics).

סיכם: אראל סגל-הלוי.