הערה כללית: את כל קובצי המקור יש להגיש תחת Package בשם test.

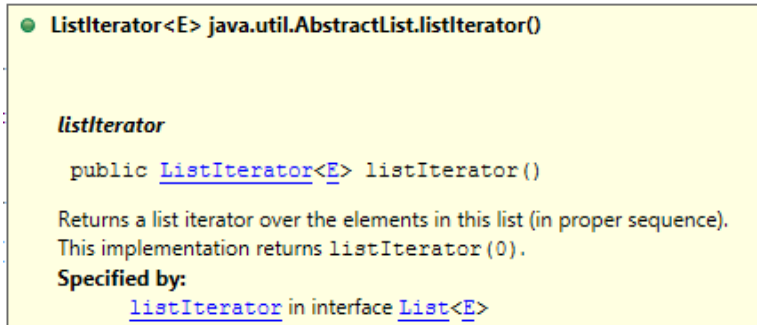
תרגיל 1 – Object Adapter (10%)

בקובץ Q1.java תמצאו מבנה נתונים חדש בשם Ring המהווה רשימה מעגלית. Ring ירשה את LinkedList. אך כדי ש Ring תתנהג כרשימה מעגלית עליה לשנות את התנהגות ה iterator המקורי. לשם כך נמצאת המחלקה RingIterator בתוך המחלקה Ring.

RingIterator תהווה Object Adapter ע"י כך שבאמצעות הכלה של ה iterator המקורי היא תממש את הממשק הנדרש של iterator. כמובן, ל RingIterator תמיד יש next. כאשר הגענו לסוף הרשימה המקושרת, הפעולה next תוביל אותנו חזרה לאיבר הראשון.

לדוגמה, ב APItest תוכלו לראות הכנסה של a,b,c,d אך מכיוון שרצנו על הרשימה עד ל size+2 הפלט כולל חזרה על שני האיברים הראשונים: a,b,c,d,a,b

טיפ: ירשתם גם מתודה בשם listIterator שמחזירה iterator חדש שעובר על כל האיברים כרשימה רגילה (לא מעגלית). במידת הצורך תוכלו להשתמש בה.



<מתאים לשבוע 2 בקורס>

תרגיל 2 – Builder ו Abstract Factory (12%)

שאלה 1:

ברצוננו ליצור את הטיפוס Goblin כאשר כל אובייקט מטיפוס זה הוא immutable. לכן, כל שדותיו מוגדרים כ final. אולם, רק השם והצבע הם שדות חובה. השלימו את הקוד (מקמות חסרים + מתודות ריקות) ע"פ ה Builder pattern עבור אובייקטים שהם immutable.

שימוש לדוגמה:

```
//new Goblin(...); // cannot create goblins without the builder
new Goblin.GoblinBuilder("zoot","green").withSize(500).withIq((byte)40).build();
Goblin g1=new Goblin.GoblinBuilder("danganesh","green").withSize(600).build();
Goblin g2=new Goblin.GoblinBuilder("gilgal","red").withIq((byte)100).build();
```

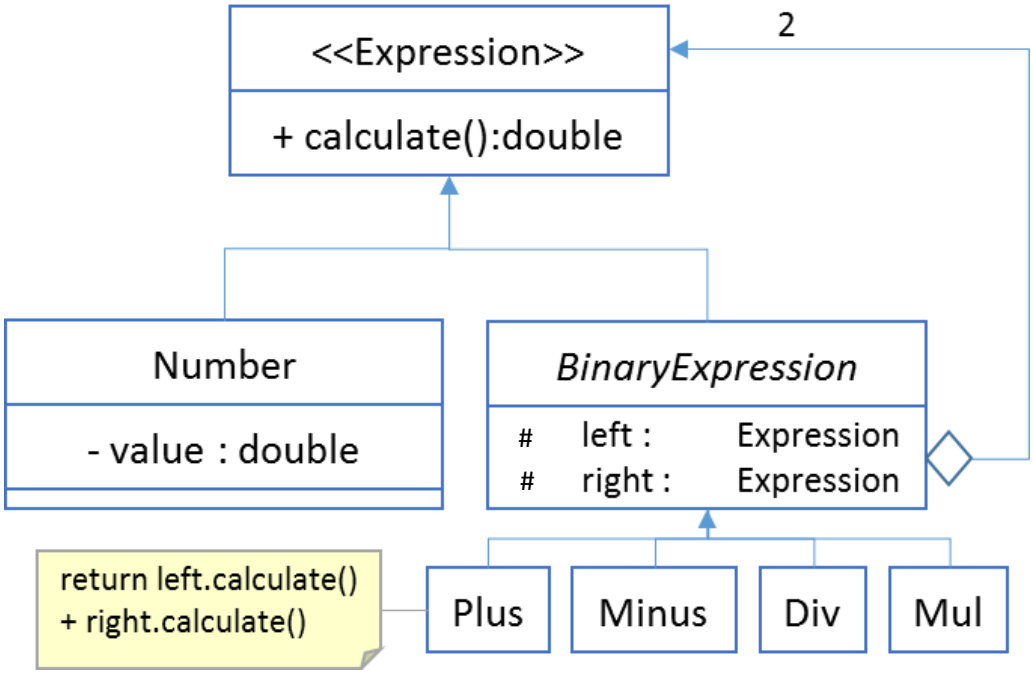
שאלה 2:

בקובץ Q2.java, עבור הטיפוסים מהסוג של A ו B, השלימו את הקוד הבא כך ש AbstractFactory1 יחזיר מצורים מסוג "1" ואילו AbstractFactory2 יחזיר מוצרים מסוג "2".

<מתאים לשבוע 4 בקורס>

תרגיל 3 – Interpreter Pattern (12%)

א. נתון תרשים ה class Diagram הבא, ממשו את הטיפוסים השונים המוצגים בו במחלקות Java ב package בשם test.



ב.

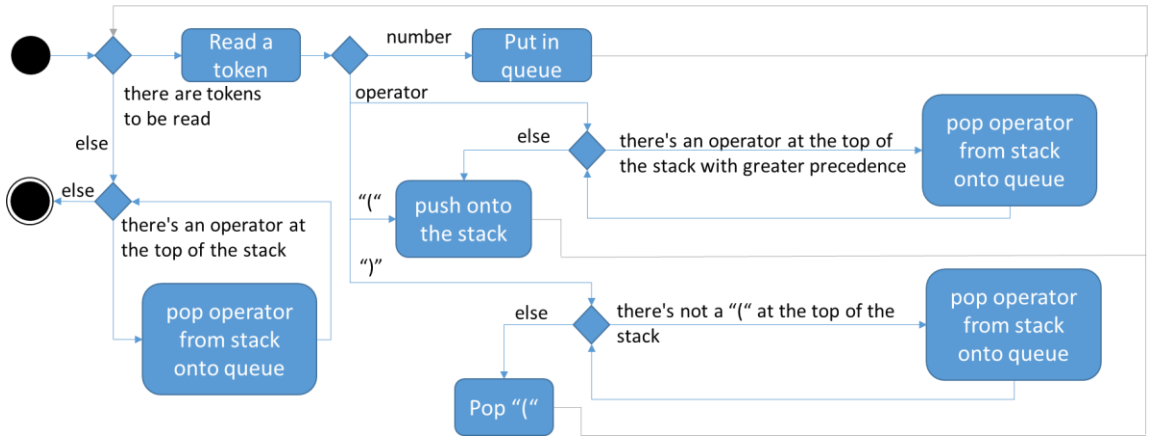
אחת הדרכים לבטא רצף של פעולות ב UML הוא ע"י activity diagram. בעצם מדובר באוטומט פשוט. העיגול השחור מהווה את נקודת ההתחלה, ואילו העיגול השחור המוקף במעגל מצין את נקודת הסיום. כל ריבוע מעוגל מציין פעילות (activity) שיש לבצע ואילו המעוין מבטא תנאי.

בהינתן מחרוזת של ביטוי, לדוג' 3+(4/2)*5, יש לפרש אותה ולחשב את התוצאה. עבור החישוב תצטרכו ליצור את האובייקטים המתאימים, לדוג':

```
Expression e=new Plus(new Number(3) , new Mul( new Div(new Number(4), new Number(2)) ,
new Number(5)));

return e.calculate();
```

כדי שתוכלו ליצור את האובייקטים המתאימים למחרוזת עליכם תחילה לפרש ולסדר אותה. לשם כך ישנו אלג' של דייקסטרה בשם Shunting-yard, המובא לפניכם כ activity diagram.



בהינתן ביטוי infix, האלגוריתם מסדר את המספרים בתור, ומשתמש במחסנית כדי להכניס את האופרטורים לתור זה בסדר שמציג את הביטוי כ postfix. למשל עבור הדוגמא לעיל בסוף האלג' התור יראה כך: +*342/5. כשנקרא את התור הפוך (כלומר, מימין לשמאל) נבין שעלינו לבצע חיבור של (הכפלה של 5 עם (חלוקה של 4 ב 2) עם 3.

לפיכך, נוכל לייצר בהתאם לביטוי את המופעים של Plus, Minus, Mul, Div, Number ולחשב את תוצאת הביטוי.

ב package בשם test, ממשו את המתודה במחלקה הבאה כך שבהינתן ביטוי כמחרוזת, תחזירו את תוצאת החישוב של הביטוי.

```
public class Q3 {
	public static double calc(String expression){
		return 0;
	}
}
```

הגשה ל ex3, יש להגיש את הקבצים הבאים:

Expression.java, Number.java, BinaryExpression.java, Plus.java, Minus.java, Div.java, Mul.java, Q3.java

<מתאים לשבוע 6 בקורס>

תרגיל 4 – תכנות מקבילי (בסיסי) + Future + Observer (12%)

כפי שלמדנו, ה Future<V> הרגיל חושף מתודה get אשר גורמת לנו להמתין אם קראנו לה לפני שהערך V הוזן ל Future ע"י ה Thread Pool. כדי לעקוף את הבעיה, עליכם לממש את המחלקה ObservableFuture אשר תעטוף Future, תמתין ברקע לערך V, וכ Observable היא תודיע לכל Observers שלה כאשר ה V הגיע. בפרט במחלקה ObservableFuture:

- הבנאי יקבל אובייקט מסוג Future (הרגיל)
- המתודה get תחזיר את הערך V ללא כל המתנה (אם הוא אינו מוכן יחזור null)
  - ההנחה היא ש observers יקראו ל get רק לאחר שקבלו נוטיפיקציה על כך ש V הגיע.

<מתאים לשבוע 9>

תרגיל 5 – Active Object, חיקוי של Future (12%)

א. **עבודה עם Future (30 נק')**

בקובץ Q1a.java עליכם לממש את המתודה threadIt כך שבהינתן פונקציה f שמחזירה ערך מטיפוס פרמטרי V, המתודה threadIt תריץ את f בת'רד נפרד ותחזיר את הערך V לתוך אובייקט מסוג Future<V>.

טיפ: מותר ואף רצוי להשתמש בספריות קוד קיימות של Java.

מוד האימון ניתן ב MainTrain1a.java

ב. **Active Object (70 נק')**

בקובץ Q1b.java עליכם לממש Active Object.

- במתודה push נזריק לו משימות מסוג Runnable.
- בת'רד נפרד ברקע, ה Active Object שלנו יריץ את המשימות בזו אחר זו (באותו הת'רד).
- באמצעות המתודה close נבצע יציאה מסודרת של ה Active Object שכוללת סגירת הת'רד שפתחנו. עליכם להריץ את כל המשימות שניתנו לפני הקריאה ל close. לאחר הקריאה ל close אין לקבל משימות חדשות.

<u>אין להיעזר בשאלה זו במשתנה מסוג ExecutorService או כל Thread Pool מוכן אחר.</u>

<מתאים לשבוע 10>

תרגיל 6 – fork-join pool (12%)

נתונה לכם המחלקה BinTree עבור ייצוג של עץ בינארי.

- מחלקה זו אינה לעריכה ואינה להגשה.
- אובייקט של BinTree מיצג קודקוד בעץ.
  - תוכלו לבצע get לערך שהקודקוד מכיל (הערך מסוג int)
  - תוכלו לבצע get לבן השמאלי ולבן הימני של הקודקוד אם הם קיימים, אחרת יחזור null.

הבדיקה יוצרת עץ בינארי מלא (כלומר כל קודקוד מכיל בדיוק 0 או 2 בנים) עם ערכים אקראיים בקודקודים.

עליכם לחפש באופן רקורסיבי את הערך המקסימאלי בעץ. אך כדי ליעל את החיפוש עליכם להשתמש ב fork join pool. בכל איטרציה החיפוש בתת העץ השמאלי יתבצע בת'רד אחר של ה fork join pool.

לשם כך עליכם לממש את המחלקה ParMaxSearcher כסוג של RecursiveTask.

בבדיקה ב MainTrain2 אנו מייצרים עץ בינארי מלא שבקודקודיו ערכים אקראיים. לאחר מכן אנו מייצרים מופע של ParMaxSearcher שמוזרק לתוך ה fork join pool. אנו בודקים ש:

- החישוב אכן מסתיים בתוך שנייה כפי שהוא אמור, אחרת הקוד נחשב כתקוע וכל ניקוד השאלה ירד
- הערך המקסימלי אכן כזה
- אכן ביצעתם שימוש ב fork join pool

<מותאם לשבוע 11>

תרגיל 7 – CompleteableFuture (12%)

בקובץ MyFuture.java עליכם לממש את המתודות, set, thenDo, finallyDo כחיקוי של CompletableFuture כך שנוכל להפעילו כבדוגמה הבאה:

```
final int sum[]={0};

MyFuture<String> mf=new MyFuture<>();

mf.thenDo(s->Integer.parseInt(s)).thenDo(x->x*2).
      thenDo(x->sum[0]+=x).
            finallyDo(x->System.out.println("result1: "+x));

mf.set("42");
```

לאחר יצירה של MyFuture ניתן לשרשר מראש פעולות של thenDo() כאשר כל אחת מהן מקבלת פונקציה כביטוי למבדה שיכולה להחזיר ערך מטיפוס שונה. בדוגמה לעיל הפעולה הראשונה ממירה String ל Integer, השנייה מכפילה אותו פי 2, השלישית מוסיפה אותו ל sum[0].

המתודה finallyDo עוצרת את השרשור. היא מקבלת ביטוי למבדה שצורך את הערך (לכשיגיע) ללא החזרה של ערך כלשהו.

כל ההגדרות הללו ניתנות מראש עוד לפני שהוזן ערך ל MyFuture.

המתודה set תזין את הערך המבוקש ל MyFuture ותגרור תגובת שרשרת של הפעלת כל הפעולות לפי הסדר שהוגדר. בדוגמה הזנת המחרוזת "42" תוביל להדפסה של result1: 84 ולשינוי ערכו של sum[0] בהתאם.

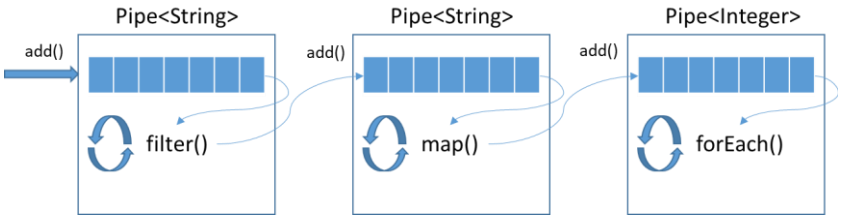
<מותאם לשבוע 12>

תרגיל 8 – חיקוי של Stream (18%)

ברצוננו לממש את המחלקה Pipe<E> (צינור). ראו את השורות הבאות ב MainTrain1.java:

```
Pipe<String> ps=new Pipe<String>();
int sum[]= {0};
// sum all lengths of strings with length under 4
ps.filter(s->s.length()<4).map(s->s.length()).forEach(x->sum[0]+=x);
```

לכל אובייקט Pipe<E> יש תור של E-ים שהוא thread safe ויכול להכיל עד 100 איברים. בדומה ל Active Object, לכל Pipe יש ת'רד אקטיבי ברקע. הת'רד פעיל רק כאשר יש נתונים בתור. הוא שולף E-ים מהתור בזה אחר זה ומבצע עליהם פעולה מוגדרת מראש. חלק מהפעולות מאפשרות העברת נתונים ל Pipe אחר. כך, ניתן להגדיר מראש "פס ייצור" של עיבוד שיחל לעבוד ברגע שהנתונים יתחילו לזרום. התרשים הבא ממחיש בצורה ויזואלית את פס הייצור שיוצר הקוד לעיל:



הפעולות הן:

- Filter – בהינתן תנאי, המתודה תעביר את כל ה E-ים שעליהם התנאי מחזיר אמת ל Pipe הבא
  - בדוגמה ps הוא Pipe<String> והתנאי הוא כל מחרוזת שאורכה קטן מ 4.
- Map – בהינתן פונקציה מ E ל R (טיפוסים פרמטריים), המתודה תמיר כל E בתור ל R ותעביר אותו ל Pipe הבא
  - בדוגמה, המתודה map הופעלה מאובייקט ה Pipe<String> שהחזירה filter. הפונקציה ממירה כל מחרוזת לאורך שלה.
- forEach – בהינתן "צרכן" של E, המתודה תצרוך כל E בתור. זו פעולה טרמינלית – אחריה לא ניתן לשרשר פעולות נוספות.
  - בדוגמה, המתודה forEach הופעלה מאובייקט ה Pipe<Integer> שהחזירה map. הצרכן צובר כל int בתור לתוך sum[0].
- Add – מכניסה אובייקט E לתור או ממתינה כל עוד התור מלא
- Stop – תעצור מידית את פעולת ה Pipe ואת ה Pipe שאחריו בהנחה והיה כזה.
  - כך, עצירה של ps תגרור עצירה של כל ה Pipe-ים.
  - מתודה זו הוגדרה בממשק Stoppable שאותה Pipe נדרשת לממש

הבדיקות ב MainTrain1:

א. לאחר הגדרת פס הייצור נוספו אך ורק 3 ת'רדים (אחד לכל Pipe שהוגדר)
ב. לאחר הכנסה של כמה מחרוזות ל ps מתקבלת התוצאה הנכונה ל sum[0]
ג. לאחר קריאה ל stop שכל הת'רדים שפתחנו נסגרו בהתאמה

הבדיקות ב MainTest1 דומות. למען הסר ספק, הקוד צריך להיות גנרי ע"פ ההגדרות ולא רק מתאים לדוגמה לעיל. בפרט, ייתכן מספר Pipe-ים שונה, סדר הפעלה שונה, פרמטרים שונים לפונקציות, איברים מסוג שונה וכמות שונה של איברים. בנוסף הקלט אקראי.

<מותאם לשבוע 13>

בהצלחה!