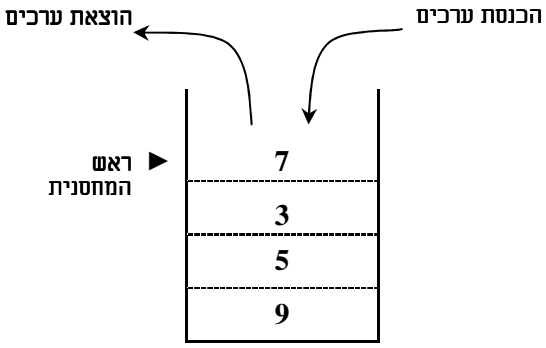
פרק 5
מחסנית

מחסנית (**Stack**) היא אוסף של ערכים המאורגן כסדרה, עם הפעולות הבאות: ניתן להכניס ערכים למחסנית ולהוציא אותם ממנה – כל זאת דרך קצה אחד, הנקרא **ראש המחסנית** (כמתואר באיור). הכנסת ערך למחסנית מוסיפה ערך חדש בראש המחסנית, מעל כל אילו המצויים כבר בתוכה. הוצאת ערך מהמחסנית, מסירה את הערך המצוי בראש המחסנית. הערך שהוכנס אחרון למחסנית הוא זה שנמצא בראש המחסנית, והוא היחידי החשוף לשימוש.

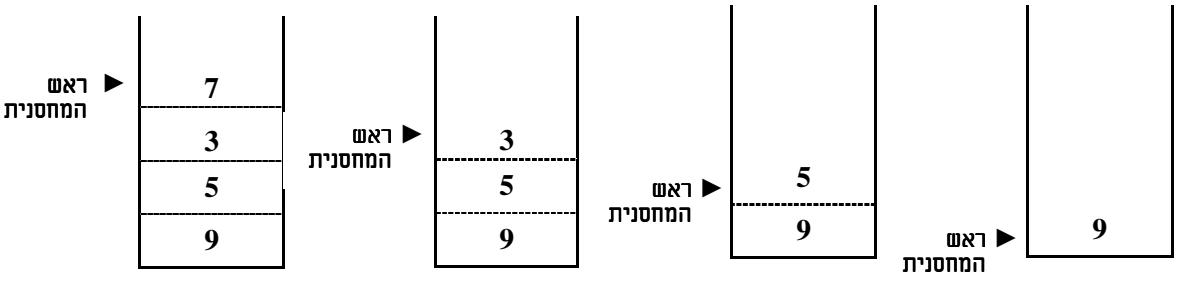
הטיפול בערכים דרך ראש המחסנית יוצר מצב שבו ערך שנכנס אחרון יוצא ראשון. גישה כזו לטיפול בערכים נקראת **LIFO** – ראשי התיבות של המילים: **L**ast **I**n **F**irst **O**ut.

הגדרת המחסנית מאפשרת הכנסה והוצאה של ערכים, כך שהמחסנית היא מבנה דינמי המשתנה ללא הרף. האיור הבא הוא "תצלום" מצב רגעי של מחסנית לאחר שהוכנסו לתוכה הערכים: 9, 5, 3, 7, בזה אחר זה (מימין לשמאל).



ניתן לראות שהערך 7, שהוכנס אחרון, נמצא בראש המחסנית.

באיור הבא ניתן לראות סדרת תצלומי-מצב של המחסנית, המתארים (מימין לשמאל) את מצבי המחסנית לאחר הכנסת כל אחד מהערכים לתוכה:



במצב הנוכחי ניתן להכניס למחסנית את הערך 5. אחרי ההכנסה יופיע ערך זה במחסנית בשני מקומות: בראשון ובריביעי (מראש המחסנית).

א. פעולות המחסנית

כדי שיהיה אפשר להוסיף ערכים למחסנית ולהוציא אותם ממנה, נגדיר את הפעולות **push** ו-**pop** (דחיפה ושליפה ממחסנית בהתאמה). הראשונה מקבלת פרמטר אחד ואינה מחזירה דבר, האחרת ללא פרמטרים ומחזירה ערך. פעולת השליפה אינה מוגדרת כאשר המחסנית ריקה, בדומה לפעולת חילוק במספרים, שאינה מוגדרת כאשר המחלק הוא אפס.

כדי לאפשר למתכנת להימנע משליפה ממחסנית ריקה, נוסיף את הפעולה **isEmpty** המחזירה 'אמת' אם המחסנית ריקה, ו'שקר' אחרת. כמובן, כדי שניתן יהיה ליצור מחסניות יש צורך בפעולה הבונה מחסנית ריקה.

מה באשר לפעולות נוספות? לעתים נרצה להציץ אל ראש המחסנית כדי לדעת מהו הערך הנמצא שם, בלי לבצע פעולת שליפה המשנה את תוכן המחסנית. לשם כך נגדיר את הפעולה **top** המחזירה את הערך העליון במחסנית, בלי לשלוף אותו. גם פעולה זו מוגדרת רק לגבי מחסנית שאינה ריקה.

ופעולה אחרונה: כמו כל עצם בג'אווה, גם מחסנית תצטרך להשתמש בפעולת **toString** מתאימה שתחזיר מחרוזת המתארת את המחסנית.
לסיכום, הפעולות המוצעות כאן כוללות את אלו שמנינו בתיאור המחסנית, שתי פעולות נוספות לנוחיות המתכנת, פעולת בניה של מחסנית וכן את הפעולה השגרתית המחזירה תיאור של העצם.

ב. מחלקת מחסנית

בשפות מבוססות עצמים משמשת המחלקה להגדרת טיפוסי נתונים שונים. המחלקה Stack מגדירה את טיפוס הנתונים מחסנית, הממשק שלה מספק לנו את הפעולות לטיפול באוסף הנתונים כפי שפורטו לעיל.

כל פעולות המחסנית שתיארנו ניתנות לביצוע על ערכים מכל טיפוס שהוא – מטיפוס בסיסי כגון מספרים או תווים, וכן מכל טיפוס עצם. בהגדרת המחלקה מחסנית יש לציין את טיפוס הערכים שיאוחסנו בה, ובכותרות הפעולות במחלקה יש לציין (כמקובל בג'אווה) את טיפוסי הערכים שהן מקבלות כפרמטרים ומחזירות כערכי החזרה. לכן, אם אנו זקוקים למחסנית לאחסון מספרים שלמים, עלינו להשתמש במחלקה המייצרת מחסניות למספרים שלמים, אם אנו זקוקים למחסנית לאחסון תווים, עלינו להשתמש במחלקה המייצרת מחסניות לתווים, וכך הלאה.

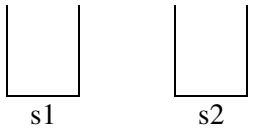
נתחיל בהצגת מחסנית המאחסנת מספרים שלמים. נקרא למחלקה IntStack.

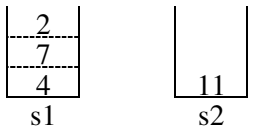
ב.1. המחלקה IntStack

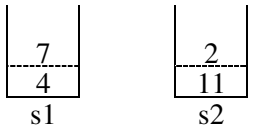
IntStack()	הפעולה בונה מחסנית ריקה
boolean isEmpty()	הפעולה מחזירה 'אמת' אם המחסנית הנוכחית ריקה, ו'שקר' אחרת
void push (**int** x)	הפעולה מכניסה את האיבר x לראש המחסנית הנוכחית
int pop()	הפעולה מוציאה את האיבר שבראש המחסנית הנוכחית ומחזירה אותו. הנחה: המחסנית הנוכחית לא ריקה
int top()	הפעולה מחזירה את האיבר שבראש המחסנית הנוכחית מבלי להוציאו. הנחה: המחסנית הנוכחית לא ריקה
String toString()	הפעולה מחזירה תיאור של המחסנית

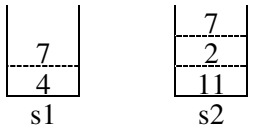
דוגמה: שימוש במחלקה IntStack

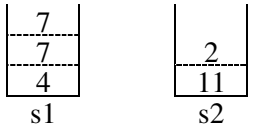
```
public static void main(String[] args)
{
      int x;
      //  יצירת שני עצמים מסוג מחסנית
      IntStack s1 = new IntStack();
      IntStack s2 = new IntStack();

      // דחיפת ערכים למחסניות
      s1.push(4);
      s1.push(7);
      s1.push(2);
      s2.push(11);

      // שליפת ערך מהמחסנית הראשונה ודחיפתו לשניה
      x = s1.pop();
      s2.push(x);

      // העתקת ערך מהמחסנית הראשונה ודחיפתו לשניה
      x = s1.top();
      s2.push(x);

      // בדיקה: אם המחסנית השניה לא ריקה, שלוף
      // ממנה ערך ודחוף אותו למחסנית הראשונה
      if (!s2.isEmpty())
          s1.push(s2.pop());
}
```











ב.2. המחלקה CharStack

אם נרצה כעת לכתוב תוכנית שאיבריה הם מטיפוס תוי, נצטרך להגדיר מחלקה חדשה בשם: CharStack. ממשק המחלקה הזו יהיה כמעט זהה לממשק המחלקה IntStack, פרט לטיפוס הפרמטר בפעולה push(…), ולטיפוס ערכי ההחזרה בפעולות pop() ו-top().

נתבונן בממשק המחלקה CharStack:

CharStack()	הפעולה בונה מחסנית ריקה
boolean isEmpty()	הפעולה מחזירה 'אמת' אם המחסנית הנוכחית ריקה, ו'שקר' אחרת
void push (**char** x)	הפעולה מכניסה את האיבר x לראש המחסנית הנוכחית
char pop()	הפעולה מוציאה את האיבר שבראש המחסנית הנוכחית ומחזירה אותו. הנחה: המחסנית הנוכחית לא ריקה
char top()	הפעולה מחזירה את האיבר שבראש המחסנית הנוכחית מבלי להוציאו. הנחה: המחסנית הנוכחית לא ריקה
String toString()	הפעולה מחזירה תיאור של המחסנית

דוגמה: שימוש במחלקה CharStack

הפעולה הראשית הבאה קולטת משפט שמסתיים בנקודה, תו אחר תו, ומדפיסה את המשפט בסדר הפוך.

```
public static void main(String[] args)
{
	char ch;
	CharStack s = new CharStack();
	IO.println ("הכנס משפט שבסיומו נקודה");
	ch = IO.readChar();
	while (ch != '.')
	{
		s.push (ch);
		ch = IO.readChar();
	}
	while (!s.isEmpty())
	{
		ch = s.pop();
		IO.print (ch);
	}
}
```

ג. שימוש במחסנית לפתרון בעיות

המחסנית משתלבת באופן טבעי בפתרון בעיות שבהן יש צורך לאחזר נתונים באופן הפוך לסדר קליטתם. להלן מספר דוגמאות:

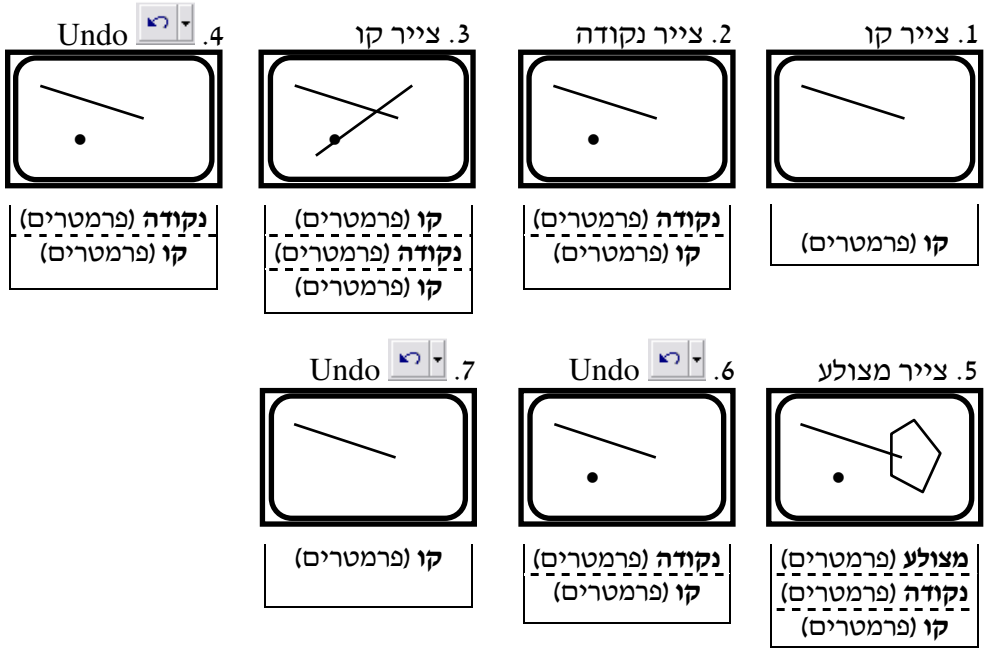
דוגמה 1: מנגנון ה-Undo (ביטול הפעולה האחרונה ב-Word)

מנגנון ה-Undo המוכר ממעבד התמלילים Word הוא הכפתור  שלחיצה עליו תבטל את הפעולה האחרונה שבוצעה. המשתמש יכול לחזור וללחוץ על כפתור זה ללא הגבלה עד לביטול הפעולה הראשונה שעשה. מנגנון ה-Undo משתמש במחסנית - בכל פעם שהמשתמש מבצע פעולה כלשהי ב-Word (כגון: כתיבה, ציור, מחיקה וכו'), נדחף למחסנית תיאור הפעולה והפרמטרים שלה. כאשר המשתמש יילחץ על כפתור ה-Undo, תישלף מראש המחסנית הפעולה האחרונה שביצע, וה-Word יבטל אותה.

לדוגמה, המשתמש ביצע ב-Word את 7 הפעולות הבאות:

1. ***צייר-קו***
2. ***צייר-נקודה***
3. ***צייר-קו***
4. ***Undo*** 
5. ***צייר-מצולע***
6. ***Undo*** 
7. ***Undo*** 

ניתן לראות את מצבם של המסך ושל מחסנית ה-Undo לאחר כל פעולה שהמשתמש ביצע:



דוגמה 2: בדיקת תקינות סוגריים בביטוי חשבוני

נגדיר ביטוי חשבוני תקין מבחינת סוגריים: ביטוי המכיל סוגריים מסוגים שונים במספר לא מוגבל, ובלבד שיהיו מאוזנים. איזון הסוגריים מחייב שמספר הסוגריים-הפותחים והסוגריים-הסוגרים יהיה שווה, וכן שכנגד כל סוגר-פותח יימצא סוגר-סוגר מאותו סוג במקום המתאים.

לדוגמה, הביטויים האלה תקינים: ((a))

(b + a - 2 * 7)

{ + 32 * (37 *) / [5 + 1]} - 4

(שימו לב: הביטוי האחרון תקין מבחינת הסוגריים, אף שכביטוי חשבוני הוא אינו תקין.)

ואילו הביטויים האלה אינם תקינים: a + ((c

([3 + a) + 4]

[ ) (5 - 3] * [2 - 3]

כאשר קיימים בביטוי סוגי סוגריים שונים, אין זה מספיק למנות את הסוגריים, צריך גם לדעת מהו סדר הופעתם. בעיה זו דומה למנגנון ה-**Undo** שעסקנו בו קודם. גם שם היה צורך לזכור את הפעולות שהופיעו ואת סדר הופעתן.

נשים לב שלגבי בדיקת תקינות סוגריים אין כלל חשיבות לכך שמדובר בביטוי חשבוני. תוכנית הבדיקה תעבוד על כל שרשרת תווים. לנוחות הדיון, נמשיך לדבר על "ביטוי".

נשתמש במחסנית שערכיה מטיפוס תו, CharStack. נסרוק את הביטוי משמאל לימין. כיוון שאנו בודקים את תקינות הביטוי רק מבחינת הסוגריים, נתעלם מכל התווים שאינם פותחים או

סוגרים. בזמן סריקת הביטוי נדחוף למחסנית כל פותח שניתקל בו, עד שנגיע לסוגר הראשון בביטוי.

מהו הערך הנמצא בשלב זה בראש המחסנית? כיוון שדחפנו למחסנית רק פותחים, בראשה נמצא עתה הפותח האחרון שבו נתקלנו. נבדוק אם הוא מתאים בסוגו לסוגר הנוכחי. אם כן, התת-ביטוי שבין הפותח לסוגר תקין, שכן לא היו בו סוגריים נוספים. נשלוף את הפותח מהמחסנית ונמשיך בבדיקת הביטוי. אם הסוגר והפותח אינם מאותו סוג, הביטוי בוודאי אינו תקין.

נמשיך בבדיקה באותו אופן; כל פותח שנקרא מהקלט יידחף למחסנית, וכל סוגר יוביל לניסיון לשלוף פותח מהמחסנית. הפותח שנשלוף הוא האחרון שלא שייכנו לשום סוגר. אם הפותח ששלפנו מהמחסנית והסוגר שקראנו מהקלט מתאימים, הם שייכים זה לזה וכל התת-ביטוי שביניהם תקין. אם אינם מתאימים, הביטוי אינו תקין והסריקה תיפסק. ניסיון לשלוף פותח ממחסנית ריקה מצביע גם הוא על ביטוי שאינו תקין, שכן אז קיימים יותר סוגרים מפותחים.

בכל שלב, אם לא התגלתה אי התאמה שגרמה להפסקת הסריקה, התת-ביטוי שבין הפותח שהוצאנו מהמחסנית לבין הסוגר המותאם לו תקין מבחינת הסוגריים. התחלנו את הסריקה עם מחסנית ריקה. אם הגענו לסופו של הביטוי מבלי שמצאנו אי התאמה והמחסנית התרוקנה, הביטוי כולו תקין מבחינת הסוגריים. כמובן שאם המחסנית אינה ריקה בסוף הסריקה הרי שהביטוי אינו תקין מבחינת הסוגריים.

להלן אלגוריתם הבודק האם ביטוי חשבוני תקין מבחינת סוגריים:

בדוק-תקינות-סוגריים (str)

{ <u>טענת כניסה:</u> האלגוריתם מקבל מחרוזת str }

{<u>טענת יציאה:</u> האלגוריתם מחזיר 'אמת' אם המחרוזת תקינה מבחינת סוגריים, ו'שקר' אחרת }

(1) התחל עם מחסנית ריקה מטיפוס תו.

(2) עבור על המחרוזת str מתחילתה ועבור כל תו ch בה שהוא סוגר או פותח, בצע:

אם ch הוא פותח, דחוף אותו למחסנית.

אחרת // זהו סוגר

אם המחסנית ריקה או שהתו הנשלף ממנה אינו תואם ל ch, החזר 'שקר'

(3) אם המחסנית לא ריקה, החזר 'שקר'.

(4) אחרת, החזר 'אמת'.

כפי שניתן לראות, האלגוריתם מסתיים באחד משלושת המקרים הבאים:

- אם בשלב מסוים הסוגר והפותח הנבחנים אינם מתאימים, הביטוי אינו תקין.
- אם הגענו לסוגר והמחסנית ריקה, הביטוי אינו תקין.
- כאשר הגענו לסוף הקלט: אם המחסנית לא התרוקנה, סימן שנותר פותח שטרם הותאם לו סוגר, והביטוי אינו תקין; אם המחסנית ריקה, הביטוי תקין.

האיור הבא מציג את פעולת האלגוריתם עבור המחרוזת הבאה: ({a} + b) * [a + (b + c])

הערה: החץ ↓ באיור מתאר את ערכו של המשתנה ch בכל שלב באלגוריתם.

	מצב המחסנית	המחרוזת הנבדקת
מצב התחלתי:		({a} + b) * [a + (b + c])
א	(	↓ ({a} + b) * [a + (b + c])
ב	{ (	↓ ({a} + b) * [a + (b + c])
ג	(	↓ ({a} + b) * [a + (b + c])
ד		↓ ({a} + b) * [a + (b + c])
ה	[	↓ ({a} + b) * [a + (b + c])
ו	([	↓ ({a} + b) * [a + (b + c])
ז	([	↓ ({a} + b) * [a + (b + c])
הסריקה הופסקה! הביטוי אינו תקין.		

נממש את האלגוריתם "בדוק-תקינות-סוגריים" בסביבת העבודה כפעולה בוליאנית שמקבלת מחרוזת. הפעולה תחזיר **true** אם המחרוזת תקינה מבחינת סוגריים, ו-**false** אחרת. פעולה זו יכולה להופיע בתוכניתנו כפעולת מחלקה, כלומר פעולה עם האפיון "סטטי" (אם אינכם זוכרים, חזרו לפרק 3 לסעיף "איברי מחלקה").

```
public static boolean isExpressionOk (String exp)
{
    char ch,old_ch;
    CharStack s = new CharStack();

    for (int i=0; i<exp.length(); i++)
   {
       ch = exp.charAt(i);
       if (ch=='(' || ch=='{' || ch=='[')
               s.push(ch);
       else
            if (ch==')' || ch=='}' || ch==']')
            {
                 if (s.isEmpty())   // יש סוגר ללא פותח
                         return false;
                 old_ch = s.pop();
                 if (ch==')' && old_ch!='(')  // סוגר ופותח לא מאותו סוג
                         return false;
                 if (ch=='}' && old_ch!='{')  // סוגר ופותח לא מאותו סוג
                         return false;
                 if (ch==']' && old_ch!='[')  // סוגר ופותח לא מאותו סוג
                         return false;
            }
     }
     // סיימנו לעבור על כל התווים במחרוזת
     // אם המחסנית לא ריקה - נותר פותח שאין לו סוגר
    return s.isEmpty();
}
```

ד. המחלקה מחסנית בשפת ג'אווה

ד.1. ייצוג המחסנית

אפשר לייצג מחסנית באמצעות מערך שבו יאוחסנו ערכי המחסנית. משתנה בשם top מטיפוס **int** יכיל את מספר התא במערך בו נמצא הערך שבראש המחסנית. כאשר המחסנית ריקה, יהיה ערכו של top שווה ל-(-1). בעת דחיפת ערך למחסנית נגדיל את ערכו של top באחד, ואז נציב את הערך החדש בתא מספר top. כאשר נרצה לשלוף ערך מן המחסנית, נחזיר את הערך במקום top, ונקטין את ערכו של top באחד.

על פי הייצוג הזה נממש את המחלקה IntStack:

```
public class IntStack
{
     public static final int STACK_MAX_SIZE = 100;
     private int[] data;
     private int top;

     public IntStack(){
          this.data = new int[STACK_MAX_SIZE];
          this.top = -1;
     }
     public boolean isEmpty()
     {
          if(this.top == -1)
               return true;
          else
               return false;
     }
     public void push(int x) {
          if(this.top < this.data.length-1)
          {
               this.top = this.top + 1;
               this.data[this.top] = x;
          }
     }
     public int pop(){
          int x;

          x = this.data[this.top];
          this.top = this.top - 1;
          return x;
     }
     public int top() {
          return this.data[this.top];
     }
     public String toString()
     {
          String str = "bottom: ";
             for (int i=0; i<=this.top; i++)
             {
                   str+=this.data[i]+" ";
             }
             str+="(top)" + "\n";
             return str;
      }
}
```

ד2.. מגבלות הייצוג

המחלקה Stack כפי שנכתבה עד לכאן, מגדירה מחסנית היכולה להכיל עד 100 איברים - מספר המוגדר על ידי הקבוע STACK_MAX_SIZE, ואיבריה הם מטיפוס מסוים.

מגבלת מקום:

מחסנית מעצם הגדרתה אינה מוגבלת לגודל כלשהו ואפשר לדחוף לתוכה מספר לא מוגבל של ערכים, ואכן פעולת הממשק ()push אינה מכילה הנחה כלשהי המגבילה את כמות האיברים במחסנית. כשאנו מייצגים מחסנית בעזרת מערך, גודלה של המחסנית מוגבל לגודל המערך. בשל הגבלה זו עלולה להתעורר שגיאה בעת ניסיון לדחוף איבר למחסנית מלאה. שגיאה זו נקראת **גלישה (overflow).** ברור שיש לתעד בממשק המחלקה Stack את המגבלה ולציין מהו המספר המקסימלי של ערכים שהמחסנית יכולה להכיל. תיעוד זה משמש כהתרעה למשתמש במחלקה על מגבלת המימוש, אך אינו מונע את השגיאה.

כיצד נוכל למנוע גלישה? אפשרות אחת היא להוסיף למחלקה Stack פעולה המחזירה 'אמת' אם המחסנית מלאה, ו-'שקר' אחרת. זהו פתרון בעייתי, שכן הוא מרחיב את הממשק על ידי פעולה שאינה נדרשת למחסנית עצמה. אפשרות נוספת היא להגדיל את המערך data בצורה דינמית על ידי יצירת מערך עזר בגודל המערך data +1 (או אף גדול יותר), העתקת כל ערכי המערך המקורי אל המערך החדש והצבת ההפניה אל המערך החדש ב-data. יש לעשות זאת בכל פעם שדוחפים ערך למחסנית. פתרון זה מגדיל את הסיבוכיות של ההכנסה והופך את הקוד למורכב יותר, אך הוא פתרון נכון כאשר ממשים את המחסנית באמצעות מערך.

בהמשך היחידה נלמד מימוש אחר של מחסנית שבו לא קיימת למעשה הגבלת מקום.

מגבלת טיפוס:

הגדרת המחסנית תלויה בטיפוס האיבר: טיפוס זה מופיע כטיפוס הפרמטר או ערך החזרה בחלק מהפעולות, וכן בטיפוס האיבר של המערך data, שבו מוחזקים איברי המחסנית. לכן עלינו לשנות את הגדרת המחלקה Stack בכל פעם שאנו זקוקים למחסנית עם טיפוס איבר אחר. לדוגמה, אם נרצה שהמחלקה Stack תגדיר מחסנית שאיבריה הם מחרוזות, יהיה צורך לשנות את הקובץ הקיים IntStack.java ולהחליף כל מופע של הטיפוס **int** בטיפוס String (סך הכול 6 החלפות. בדקו זאת!). החלפה כזו תפתור את הבעיה כל זמן שהתכנית שלנו משתמשת במחסנית אחת או שכל המחסניות בתוכנית הן בעלות אותו טיפוס איבר. ומה אם נרצה להשתמש בתוכנית בשתי מחסניות שטיפוס איבריהן שונה? במקרה שכזה נצטרך לכתוב שתי מחלקות שונות, בעלות שמות שונים, כאשר ההבדל היחיד ביניהן הוא טיפוס האיברים. למשל אם נרצה באותה תוכנית להשתמש במחסנית של מחרוזות ובמחסנית של תווים, נצטרך להגדיר שתי מחלקות: CharStack.java שבה טיפוס האיברים הוא char, ו-StringStack.java שבה טיפוס האיברים הוא String.

ה. גנריות

את מגבלת הטיפוס ניתן לפתור באמצעות מנגנון ה**גנריות** (**genericity**). מנגנון זה מאפשר להגדיר מחלקה מבלי לקבוע בהגדרה את טיפוס האיבר שיאוחסן בה. רק כאשר משתמשים בהגדרת המחלקה, ליצירת עצם או להכרזת טיפוס של משתנה, יש לציין את הטיפוס המדויק של האיבר. כלומר, מאותה מחלקה גנרית ניתן ליצור מחלקות מחסנית קונקרטיות בעלות טיפוסי איבר שונים.

כך נראית כותרת המחלקה הגנרית של מחסנית:

```
public class Stack<T>
{
    מימוש המחלקה כאשר הטיפוס הוא T (המימוש יתואר בהמשך)
}
```

הסימן T כאן אינו טיפוס מסוים, אלא "מחזיק מקום" (place holder) לטיפוס שעדיין לא נקבע. התוספת <T> אחרי שם המחלקה משמעותה, שזו הגדרת מחלקה גנרית שבה הטיפוס, שעדיין לא נקבע, מסומן באות T. (כפי שנראה בהמשך יש שימוש ב-T בתוך קוד המחלקה). אין חשיבות לאות T, ניתן לסמן את הטיפוס בכל אות או מזהה אחרים. בעת השימוש במחלקה, יש לקבוע מהו הטיפוס T.

למשל, כשניצור מחסנית של מחרוזות: `Stack<String> s = new Stack<String>();`

כאן השתמשנו במחלקה פעמיים: להכרזת טיפוס המשתנה s וכן ליצירת עצם חדש מן המחלקה. בשני השימושים השתמשנו בשם המחלקה ואחריו <String> . המשמעות היא שאנו משתמשים כאן במחלקה, כאשר T נקבע להיות הטיפוס String.

<u>הערה</u>: אפשר להסתכל על T כ**משתנה טיפוס**. הסוגריים סביב T אינן (), כדי ליצור הבדל ברור מפונקציות רגילות במתמטיקה ומסגנון כתיבת פעולות עם פרמטרים.

<u>הערה לגבי השימוש במנגנון הגנריות</u>: אפשר לקבוע את T להיות טיפוס של עצם כל שהוא, אך לא להיות טיפוס בסיס (primitive), כגון **int**. (מגבלה זו נובעת ממימוש רעיון הגנריות, אך נושא זה רחב ולא נעסוק בו כאן). בהמשך נראה שלמרות מגבלה זו ניתן לייצר מחסנית של מספרים שלמים וכן מחסנית של כל טיפוס בסיסי ולהשתמש בה ללא קושי.

ה.1. שימוש במחלקה הגנרית

ניתן ליצור מחסניות שאיבריהן מטיפוס עצם, כאשר בעת היצירה מציינים את הטיפוס הרצוי. יותר מזה, ניתן ליצור באותה תוכנית מספר מחסניות מטיפוסים שונים:

```
public static void main(String[] args)
{
        Stack<Point> p = new Stack<Point>();
        Stack<String> s = new Stack<String>();
        …
        p.push (p1);
        s.push (“adam”);
}
```

כאמור לעיל, הטיפוס חייב להיות טיפוס של עצם. דרישה זו מעלה בעיה: כיצד נייצר מחסנית שאיבריה מטיפוס פשוט, כגון **int**? לשם כך נשתמש במנגנון נוסף הקיים בשפה.

לכל טיפוס פשוט, מוגדרת מחלקה מקבילה: Integer עבור **int**, Double עבור **double**, Char עבור **char**, וכן הלאה. מחלקות אלו מאפשרות להתייחס לערכים של טיפוסים בסיסיים כאל עצמים. מחלקות אלו נקראות **מחלקות עוטפות (type wrapper classes)**. (מידע נוסף עליהן ניתן למצוא בתיעוד on-line של השפה). כדי להקל על המתכנת, ההמרות בין ערכים מטיפוסי בסיס לעצמים עוטפים ובחזרה נעשות אוטומטית על ידי המערכת, בהתאם לצורך, והמתכנת אינו נדרש לכתוב המרות בתכניתו. (**אזהרה**: המרה אוטומטית כמתואר כאן קיימת בג'אווה רק מגרסה 1.5.)

כדי לייצר ולהשתמש במחסנית של מספרים שלמים, מגדירים מחסנית מטיפוס Integer. ניתן להכניס למחסנית ולהוציא ממנה הן עצמים מטיפוס Integer, והן ערכים מטיפוס **int**:

```
public static void main (String[] args)
{
        Stack<Integer> s = new Stack<Integer>();
        s.push (101);
        int a = s.pop();
}
```

בשורה הראשונה בגוף הפעולה יצרנו מחסנית מטיפוס Integer. בשורה השנייה הכנסנו אליה איבר 101 שהוא ערך מטיפוס **int**. התבצעה המרה אוטומטית לעצם מטיפוס Integer (המערכת יודעת שקיים צורך בהמרה כיון שהארגומנט של הפעולה הוא מטיפוס **int** ואילו טיפוס האיבר של המחסנית הוא Integer). בשורה השלישית הוצאנו את האיבר והכנסנו אותו למשתנה a. שוב התבצעה המרה אוטומטית, הפעם מעצם מטיפוס Integer לערך מטיפוס **int**.

ה.2. מימוש המחלקה הגנרית

ראינו כבר את הגדרת הכותרת של המחלקה הגנרית, נדון עתה בתכולתה. התכונות top, ו-STACK_MAX_SIZE אינן משתנות.

```
public class Stack<T>
{
    public static final int STACK_MAX_SIZE = 100;
    private int top;
```

עתה עלינו להגדיר כתכונה, מערך מטיפוס T:

```
    private T[] arr;
```

במקום הטיפוס מופיעה האות T, הפעם ללא הסוגריים המשולשים שבהגדרת המחלקה. הסוגריים אינם מופיעים בתוך מימוש המחלקה, אלא רק בכותרתה ובעת השימוש בה.

במימוש הפעולות הרגילות של המחלקה, השינוי קל מאוד: במקום טיפוס האיבר מופיעה האות T. להלן הגדרת הפעולה push (השינויים ב-pop וב-top דומים; ב-isEmpty אין כלל שימוש בטיפוס האיבר, ב-toString() יתבצע שרשור המחרוזות המייצגות את איברי המחסנית. במידה והאיברים הם עצמים, הרי שעבור כל אחד מהם מוגדר toString() מתאים והוא זה שישורשר, בלי שאנו נצטרך להתייחס מפורשות לטיפוס האיבר):

```
public void push(T x)
{
    if (this.top < STACK_MAX_SIZE-1)
    {
        this.top = this.top + 1;
        this.data[this.top] = x;
    }
}
```

איתחול המערך arr נעשה בפעולה הבונה, וכאן השינוי מורכב יותר. כיוון שהמערך המשמש לאחסון איברי המחסנית הוא מערך גנרי, אתחולו נעשה באופן מיוחד:

```
public Stack()
{
    this.arr = (T[]) new Object[STACK_MAX_SIZE];
    this.top = -1;

}
```

נסביר: Object היא מחלקה נתונה בג'אווה. כל עצם שנוצר ממחלקה כל שהיא, נחשב *גם* לעצם של מחלקה זו, ללא תלות במחלקה ממנה נוצר. במקרה שלנו, השימוש ב-T[] arr בהגדרת

המחלקה הגנרית קובע, כי המשתנה arr עתיד להפנות למערך שיחזיק הפניות לעצמים, אך טיפוס העצמים עדיין לא נקבע וגם המערך עצמו עדיין לא נוצר. כאשר משתמשים במחלקה הגנרית ליצירת עצם חדש, נקבע ל-T טיפוס קונקרטי, מופעלת הפעולה הבונה, ובה נוצר ומופנה על ידי המשתנה arr, מערך בגודל STACK_MAX_SIZE של הפניות לעצמים מטיפוס Object. אך אנו מעוניינים בעצמים מטיפוס T ולא מטיפוס Object! הביטוי (T[]) הוא **המרת טיפוס**, cast. הוא מכריז כי כל העצמים במערך הם מהטיפוס שייקבע במקום ההגדרה T.

כאשר משתמשים בהגדרת המחלקה כך: **new** Stack<String>(), אזי הטיפוס String מחליף את T, בכל מקום בו הוא מופיע. אנו המתכנתים איננו רואים את ההחלפה. ההחלפה נעשית בפרט בפעולה הבונה: ההמרה (T[]) מוחלפת להמרה (String[]) ולכן נרשם במערכת כי בעצם שנוצר, המערך מחזיק הפניות לעצמים מטיפוס String.

נציין שתהליך ההגדרה של מערך גנרי פשוט יותר מבחינת הכתיבה ב-C#. ההבדל אינו מייצג רשלנות של מתכנני ג'אווה, אלא השקפות שונות על תכנון שפה, אך ההסבר לכך דורש הבנה לעומק של מנגנונים נוספים ואין מקומו כאן.

ו. פעולות נוספות למחסנית

לפעמים אנו מעוניינים לממש בסביבת העבודה פעולות נוספות המתייחסות למחסנית. היכן נממש את הפעולות הללו, האם נרחיב את הממשק המוגדר או שמא נמנע מכך? כיצד נגדיר פעולה שכזו מחוץ לממשק המחסנית? נבחן את הדוגמה הבאה.

ו.1. הפעולה ספור-ערכים

ברצוננו לממש בסביבת העבודה את הפעולה **ספור-ערכים** שתחזיר את מספר הערכים במחסנית מסוימת. עומדות לפנינו שתי אפשרויות:

1. מימושה כ**פעולה פנימית** של המחלקה Stack המופיעה בפרק זה, כלומר הוספת עוד פעולה לממשק המחלקה:

```
public int countItems()
{
    return this.top+1;
}
```

בכך אנו מרחיבים למעשה את אוסף פעולות הממשק של מחסנית, מעבר למינימום המגדיר תפקוד תקין של מחסנית.

הפעולה הפנימית, פועלת על המחסנית הנוכחית ולכן מותר לנו להשתמש בייצוג כדי לספור את מספר הערכים שבתוכה.

לרוב איננו מעוניינים להרחיב את הממשק של טיפוס נתונים עבור כל בעיה פרטית בה נתקלים ולכן עומדת בפנינו אפשרות נוספת.

2. מימושה של הפעולה כ**פעולה חיצונית** למחלקה Stack. הגדרת פעולה חיצונית פירושה למעשה שהפעולה מוגדרת וממומשת מחוץ לממשק המחסנית, במחלקה אחרת. הפעולה אינה פועלת על מופע מסוים של המחלקה בה היא מוגדרת, אלא על מופע של המחלקה מחסנית שאותו היא מקבלת כפרמטר.

נבחן מספר נקודות ונגבש את אופן הגדרת הפעולה החיצונית הרצויה לנו:

i. מכיוון שהפעולה היא מעין פעולת שירות לגבי מחסנית, היא תוגדר כפעולת מחלקה של מחלקת שירות כלשהי, בתוספת האפיון **static** (היזכרו בפרק 3 בסעיף איברי מחלקה).

ii. מאחר והפעולה ממומשת מחוץ להגדרת המחלקה ניתן להשתמש בפעולות ממשק המחסנית בלבד כדי לספור את הערכים שבתוכה.

iii. פעולה זו אפשרית לביצוע עבור מחסנית מטיפוס כלשהו מכיוון שבכל מחסנית ניתן לספור את מספר האיברים. צורת הכתיבה של פעולה חיצונית גנרית, מורכבת מעט ולכן נעדיף להגדיר פעולות חיצוניות על טיפוסים קונקרטיים בלבד ולפשט את דרך הכתיבה.

iv. מובן שאסור לנו לשנות את המבנה המקורי של המחסנית הנשלחת אל הפעולה מכיוון שסביר ביותר שנצטרך להמשיך ולהתייחס אל המחסנית בהמשך התוכנית, מעבר לפעולה בודדת זו. מכיוון שאופי החישוב מחייב שינוי הסדר של איברי המחסנית וכניסה לעומקה, נצטרך בפעולות מן הסוג הזה להעביר ("לשפוך") את איברי המחסנית למחסנית עזר שעליה ייעשו החישובים ולדאוג מיידית לבניית המחסנית המקורית מחדש.

נראה אם כן את המימוש הבא הסופר את הערכים השמורים במחסנית של שלמים:

```
public static int countItems (Stack<Integer> s)
{
     int counter = 0;
     Stack<Integer> tmp = new Stack<Integer>();

     while(!s.isEmpty())
     {
          tmp.push(s.pop());
          counter++;
     }

     while(!tmp.isEmpty())
          s.push(tmp.pop());

     return counter;
}
```

2.1. הפעולה סכום-ערכים

ננסה להגדיר עכשיו פעולה נוספת על מחסנית, פעולה הסוכמת את **ערכי** האיברים במחסנית. (למשל, אם איברי המחסנית הם 1, 6, 18 אזי הפעולה תחזיר 25). פעולה זו אינה מתאימה למחסנית בעלת איברים שלא ניתנים לסכימה. למשל, לא ניתן לסכום איברים שהם צבים או קוביות. לכן פעולה זו אינה יכולה להיות פנימית, שכן היא לא מתאימה לכל טיפוס של איברים.

ננסה להגדיר אותה אם כן כפעולה חיצונית. כמו הפעולה הקודמת (ספור-ערכים), תוגדר פעולה זו במחלקת שירות כפעולה סטטית, תשתמש רק בפעולות הממשק של המחסנית, תפעל על מחסנית המתקבלת כפרמטר ולא תוכל לפגוע במבנה מחסנית זו.
שלא כמו בפעולה הקודמת, הפעולה כלל לא יכולה להיות מוגדרת כפעולה חיצונית גנרית עבור כל טיפוס של מחסנית. הפעולה מתאימה רק לטיפוסים מסוימים, שאת איבריהן אפשר לסכום ולכן ברור שתוגדר על מחסנית קונקרטית בלבד. נראה את המימוש המתקבל עבור מחסנית של שלמים:

```
public static int  sumItems (Stack<Integer> s)
{
     int counter = 0;
     Stack<Integer> tmp = new Stack<Integer>();

     while(!s.isEmpty())
     {
          int t = s.pop();
          tmp.push(t);
          counter+=t;
     }
     while(!tmp.isEmpty())
          s.push(tmp.pop());

     return counter;
}
```

שימו לב כי הפרמטר של הפעולה הוא מטיפוס Stack<Integer> וכך גם טיפוס המחסנית המקומית tmp.

ההחלטה באיזה פתרון נבחר לצורך מימוש פעולה נוספת על אוסף נתונים, האם נגדיר בממשק פעולה פנימית נוספת או נגדיר פעולה חיצונית תעשה בהתאם לצרכים הספציפיים של הבעיה ובהתאם להנחיות אותן נקבל. דבר אחד נקבל על עצמנו כבר עכשיו באופן מחייב לכל היחידה. הגדרת פעולות חיצוניות על אוספי נתונים תעשה **רק** לגבי אוספים שבהם הטיפוס מוגדר ממש (קונקרטי). למרות שחלק מהבעיות יכולות להיפתר עבור טיפוס גנרי כלשהו, אנו נעדיף שלא להיכנס לפרטים שמעבר לתחום של יחידת לימוד זו. אי לכך פעולות חיצוניות על כל אוספי הנתונים תוגדרנה ללא שימוש במנגנון הגנריות.

ז. המחסנית - טיפוס נתונים מופשט

הבה נסכם את תכונותיה של המחסנית ואת הפעולות שלה, תוך הדגשת הקשרים בין הפעולות וההגבלות על הפעולות השונות.

תיאור המחסנית:

1. המחסנית היא אוסף של ערכים (או עצמים); האוסף יכול להיות ריק (כמוגדר בנקודה 3 להלן).
2. יש פעולה להוספת ערך למחסנית, ויש פעולה להוצאת ערך מהמחסנית. פעולת ההוצאה מחזירה תמיד את הערך האחרון שהוכנס למחסנית ועדיין לא הוצא.
3. המחסנית ריקה בדיוק כאשר כל ערך שהוכנס אליה הוצא ממנה (זה כולל את המקרה שעדיין לא הוכנס אף ערך).
4. פעולת ההוצאה אינה מוגדרת כאשר המחסנית ריקה.

תיאור זה מגדיר למעשה את המושג מחסנית. המימוש שבחרנו למחלקה מחסנית נאמן להגדרה זו, במילים אחרות, זהו מימוש נכון.

נוכיח זאת על ידי השוואה בין מימושי הפעולות לתיאור הפעולות והקשרים ביניהן.

בתכונה 2 נאמר שפעולת ההוצאה מחזירה תמיד את הערך האחרון שהוכנס למחסנית ועדיין לא הוצא. במימוש שבחרנו הערכים שעדיין לא הוצאו נמצאים במערך במיקומים מ-0 עד top. האחרון ביניהם שהוכנס, נמצא בתא שמספרו top. זהו הערך שיישלף בפעולת ההוצאה. מכאן שתכונה זו מתקיימת.

קל לראות כי גם התכונות 3 ו-4 מתקיימות במימוש שהצגנו. מכאן שהמימוש מקיים את כל ההגבלות הקיימות לגבי הפעולות שבתיאור המחסנית.

אמנם יש במימוש פעולות נוספות (למשל top()), אך אין בכך כל רע, כיון שפעולה זו אינה פוגעת בקשרים בין הפעולות האחרות.

הוכחנו נכונות המימוש. עלינו לבחון עוד נקודה חשובה והיא רמת ההסתרה של הייצוג שמספקת הגדרת המחלקה. ייצוג המחסנית הוא על ידי מערך ומצביע מיקום top. המערך מוגדר כתכונה פרטית, וכן המיקום top, לכן, המשתמש במחלקה אינו יכול לפעול ישירות על המערך או על top, אלא רק על ידי שימוש בפעולות הממשק של המחלקה. הפרדה זו בין הממשק לייצוג מבטיחה שהמגבלות בתיאור המחסנית יתקיימו תמיד, שכן הוכחנו את נכונות הפעולות. בנוסף לכך, העובדה שהגישה לתכונות הפרטיות נעשית על ידי שימוש בפעולות בלבד, מבטיחה שניתן יהיה להחליף את דרך ייצוג המחסנית ובהתאמה את אופן מימוש המחלקה בייצוג ומימוש אחר. אם המימוש החדש יציע אותן פעולות, ויקיים אף הוא את המגבלות הנדרשות, אזי כל תכנית המשתמשת במחסנית תוכל להשתמש במימוש החדש ללא כל שינוי. ואמנם בפרק הבא אנו מציעים מימוש אחר למחסנית שאינו סובל ממגבלת המקום שתוארה לעיל.

כאשר מחלקה מיישמת מפרט של טיפוס נתונים כך שפעולותיה מקיימות את המגבלות שבתיאור הטיפוס, ואין למשתמש בה כל אפשרות להשתמש ישירות בייצוג שנבחר עבור הטיפוס, אזי אנו

אומרים שהמחלקה מגדירה **טיפוס נתונים מופשט, ADT (abstract data type)**. מחלקת המחסנית היא אם כך מחלקה המגדירה טיפוס נתונים מופשט.

את הדיון בחסרונות של מחלקה שאינה מהווה טיפוס נתונים מופשט נשאיר לפרק הבא.

ח. סיכום

- ה**מחסנית** היא אוסף של ערכים המאורגן כסדרה, שהפעולות העיקריות עליו הן דחיפת ערך, ושליפת הערך האחרון שנדחף שעדיין מצוי במחסנית. גישה כזו לערכים נקראת **LIFO** – ראשי התיבות של המילים: **L**AST **I**N **F**IRST **O**UT.
- המחסנית משתלבת באופן טבעי בפתרון בעיות שבהן יש צורך לאחזר נתונים באופן הפוך לסדר קליטתם.
- **ממשק המחסנית הגנרית**:

Stack()	הפעולה בונה מחסנית ריקה
boolean isEmpty()	הפעולה מחזירה 'אמת' אם המחסנית הנוכחית ריקה, ו'שקר' אחרת
void push (T x)	הפעולה מכניסה את הערך x לראש המחסנית הנוכחית
T pop()	הפעולה מוציאה את הערך שבראש המחסנית הנוכחית ומחזירה אותו. הנחה: המחסנית הנוכחית לא ריקה
T top()	הפעולה מחזירה את הערך שבראש המחסנית הנוכחית מבלי להוציאו. הנחה: המחסנית הנוכחית לא ריקה
String toString()	הפעולה מחזירה תיאור של המחסנית

- ניתן לייצג מחסנית באמצעות מערך, אך יש לזכור כי ייצוג זה מגביל את גודל המחסנית ועקב כך יכולות להיגרם שגיאות בזמן ריצה.
- ניתן להגדיר מחסנית גנרית. במחסנית זו עדיין לא נקבע הטיפוס של איבריה. טיפוס המחלקה יקבע רק בעת השימוש במחלקה, למשל בעת יצירת עצם. ממחסנית גנרית אפשר ליצור מחסניות מטיפוסים שונים ללא צורך בשינוי המחלקה.
- המחלקה Object היא מחלקת-על של כל המחלקות (הקיימות מראש או המוגדרות בתכנית). כל עצם שנוצר ממחלקה כלשהי הוא גם מהטיפוס Object. הפנייה לעצם מטיפוס לא ידוע היא

למעשה פנייה לעצם מטיפוס Object. ניתן להמיר טיפוס של עצם מטיפוס Object לטיפוס עצם אחר C, ובלבד שאנחנו יודעים בוודאות שאמנם מדובר בעצם שנוצר מהמחלקה C.

- מחלקה עוטפת מאפשרת להשתמש בערך מטיפוס בסיס כאילו הוא עצם. בדרך כלל, ההמרה בין שתי הצורות – ערך ועצם – נעשית אוטומטית בעת הצורך.
- כדי להרחיב את הפעולות המוגדרות על מחסנית יש להחליט אם להגדיר פעולות פנימיות המרחיבות את הממשק המקורי או להגדיר פעולות חיצוניות המקבלות את המחסנית כפרמטר ועובדות עליה.
- פעולות חיצוניות על אוספים שיוגדרו ביחידה זו יטפלו רק בטיפוסים קונקרטיים ולא בטיפוסים גנריים.
- המחסנית כפי שהוצגה בפרק זה היא טיפוס נתונים מופשט.

מושגים

גלישה	overflow
גנריות	genericity
דחיפה	push
מחלקה עוטפת	type wrapper class
מחסנית	Stack
מחסנית ריקה	empty stack
נכנס אחרון יוצא ראשון	Last In First Out
שליפה	pop

תרגילים

שימוש במחסנית

מטרות כלליות

תרגול **השימוש** בפעולות הממשק של טיפוס הנתונים מחסנית.

כל השאלות בחלק זה הן על מחסנית שאינה גנרית.

שאלה 1

תארו את תכולת המחסניות s1 ו-s2 לאחר כל סדרת פעולות:

א.

```
CharStack s1 = new CharStack();
CharStack s2 = new CharStack();
s1.push('א');
s1.push('ב');
s1.push('ג');
s1.push('ד');
char ch = s1.pop();
s2.push(ch);
ch = s2.pop();
s1.push('ה');
```

ב.

```
IntStack s1 = new IntStack();
IntStack s2 = new IntStack();
s1.push(1);
s2.push(2);
s1.push(3);
s2.push(4);
s1 = s2;
s1.push(5);
s2.push(6);
```

שאלה 2

א. קבעו עבור כל אחד מהביטויים החשבוניים הבאים, האם הוא תקין מבחינת הסוגריים:

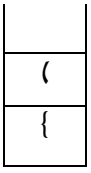
- {(a + b) + c } * [2 * {5 - a}]
- q - (r - (s + 2)) * q)
- [a +]] * [b - {a * 3}] + (a -(i)
- {1 + (a *4)} - ((2 + (3 -[5 +b])) / 2)

ב. עקבו אחרי ריצת הפעולה `isExpressionOk` לבדיקת תקינות סוגריים המופיעה בפרק עבור כל אחת מהדוגמאות שבסעיף א. הראו את מצב המחסנית בכל שלב. אפיינו את הבעיה בכל אחת מהדוגמאות שאינן תקינות.

שאלה 3

נתון הביטוי : {a + (b * c - d * [e + f]) / g}.

הביטוי ניתן כקלט לאלגוריתם שבודק תקינות סוגריים. בשלב מסוים בריצת האלגוריתם התקבל מצב המחסנית הזה :



באילו מהמקרים הבאים היה יכול להתקבל מצב המחסנית המתואר?

א. לאחר קליטת קטע הביטוי : $\{a+(b*c$.

ב. לאחר קליטת קטע הביטוי : $\{a+(b*c-d*[e$.

ג. מצב כזה לא היה יכול להתקבל בשום שלב.

ד. לאחר קליטת קטע הביטוי : $\{a+(b*c-d*[e+f]$.

שאלה 4

א. כתבו אלגוריתם הקולט סדרה של תווים, תו אחר תו, וקובע האם המחרוזת שנקלטה היא מהצורה xZyZx כאשר :

- x היא מחרוזת לא ריקה באורך לא ידוע, המורכבת מהתווים : a, b ו-c.
- y היא מחרוזת המכילה אותם תווים כמו x, רק בסדר הפוך (y הוא ההיפוך של x).
- Z הוא התו Z.

לדוגמה המחרוזות האלו הן מהצורה xZyZx : aZaZa

aacbaZabcaaZaacba

ואילו המחרוזת הזו אינה מהצורה xZyZx : aZbZa

רמז : אפשר להשתמש ביותר ממחסנית אחת.

ב. ממשו את האלגוריתם שכתבתם בסעיף א. הניחו שהקלט מסתיים בהכנסת התו enter שערכו התוי הוא 13 וניתן גם להשוות אותו כתו מול הצירוף : '\n'.

שאלה 5

א. כתבו אלגוריתם בשם **היפוך-מותנה** הקולט מהמשתמש סדרה של תווים. האלגוריתם מדפיס את התווים בסדר בו נקלטו, עד שמופיע התו '@'. תו זה אינו מודפס, אבל גורם להיפוך סדר ההדפסה של התווים בינו ובין ה-'@' הבא. יש להניח שסידרת התווים שנקלטה מכילה מספר זוגי של '@'.

לדוגמה, עבור כל אחד מהקלטים הבאים:

@ven@@im re@nd

ne@m rev@ind

יתקבל הפלט:

never mind

ועבור הקלט:

se@tp@emb@re@ so@gn@

יתקבל הפלט:

september song

ב. ממשו את האלגוריתם **היפוך-מותנה** כפעולה חיצונית. הניחו שהקלט מסתיים לאחר שהמשתמש הקיש Enter (שערכו המספרי הוא 13 או שווה לערך התוי '\n').

תרגילי מעקב

מטרות כלליות

מעקב והבנה של קטעי קוד המשתמש בטיפוס הנתונים מחסנית.

<u>שאלות על מחסנית שאינה גנרית</u>

שאלה 6

נניח כי הפעולה הבאה הוספה כפעולה פנימית של המחלקה IntStack:

טענת כניסה: הפעולה מקבלת מחסנית s שערכיה מספרים שלמים.

טענת יציאה: ???

```
public boolean secret(IntStack s)
{
	while(!this.isEmpty() && !s.isEmpty())
	{
		if(this.pop() != s.pop())
			return false;
	}
	if(!this.isEmpty() || !s.isEmpty())
		return false;
	return true;
}
```

א. תנו דוגמה לשתי מחסניות s1,s2 עבורן הזימון `s1.secret(s2)` יחזיר **false**, ודוגמה נוספת שתחזיר **true**.

ב. מה היא טענת היציאה של הפעולה secret?

ג. אילו שינויים יש לבצע במימוש הפעולה `secret` כדי שניתן יהיה להשתמש בה למחסניות מסוג מחרוזת String? נמקו תשובתכם.

שאלה 7

נתונה הפעולה החיצונית הבאה, המשתמשת במחסנית של מספרים שלמים:

```
public static int mystery(int num)
{
	int x = 0, y = 1;
	IntStack s = new IntStack();

	while(num != 0)
	{
		s.push(num % 10);
		num = num / 10;
	}
	while(!s.isEmpty())
	{
		x = x + y*s.pop();
		y = y * 10;
	}
	return x;
}
```

א. מהו המספר שיוחזר עבור הזימון `mystery(2047)`?

ב. השלימו את טענת היציאה של הפעולה `mystery`.

ג. האם ניתן לממש את הפעולה `mystery` ללא שימוש במחסנית? אם כן – כיצד? אם לא – נמקו מדוע לא.

שאלה 8

נתונות שתי מחסניות st1 ו-st2, של מספרים שלמים. המחסנית st2 ריקה, ואילו st1 מכילה מספר לא ידוע של ערכים. נתון קטע הקוד הבא:

```
while(!st1.isEmpty())
{
	x = st1.pop();
	if (!st1.isEmpty())
	{
		y = st1.pop();
		st2.push(y);
	}
	else
		st2.push(x);

	st2.push(x);
}
while(!s2.isEmpty())
{
	x = st2.pop();
	IO.print(x);
}
```

א. בהנחה שהמחסנית st1 מכילה את סדרת הערכים הבאה: 8 2 1 7 3 (8 נמצא בראש המחסנית) כתבו מה יהיה הפלט שיתקבל בעקבות הרצת קטע הקוד.

<u>שאלה על מחסנית גנרית</u>

שאלה 9

נתונה התוכנית הבאה המשתמשת במחלקה Stack:

```
public class Test
{
	public static void main(String[] args)
	{
		Stack<String> st1 = new Stack<String>();

		st1.push("i");
		st1.push("hello");
		st1.push("here");
		st1.push("world");
		st1.push("am");
		doSomething(st1);
		while(!st1.isEmpty())
			IO.println(st1.pop());
	}

	public static void doSomething(Stack<String> st2)
	{
		Stack<String> st3 = new Stack<String>();

		while(!st2.isEmpty())
		{
			String str = st2.pop();
			if(str.length() > 4)
				st3.push(str);
		}
		while(!st3.isEmpty())
			st2.push(st3.pop());
	}

}
```

עקבו אחרי ריצת התוכנית, וכתבו את הפלט.

תרגילי מימוש

מטרות כלליות

מימוש של פעולות פנימיות וחיצוניות המוגדרות על מחסניות.

<u>שאלות על מחסנית שאינה גנרית</u>

שאלה 10

נניח כי הפעולה top() לא קיימת בממשק המחסנית. כתבו פעולה חיצונית המבצעת את הפעולה הזו על מחסנית של מחרוזות (StringStack). חשבו מהו הפרמטר שמקבלת הפעולה ומה ערך ההחזרה שלה.

שאלה 11

לעיתים קרובות אנו מעוניינים לשכפל מחסנית, כלומר ליצור העתק מדויק של מחסנית קיימת. בתרגיל זה עליכם להוסיף למחלקה CharStack, פעולה-בונה-מעתיקה.

שאלה 12

ממשו את הפעולה הנמצאת במחלקה IntStack וכותרתה היא:

```
public boolean existsIn (int x)
```

טענת כניסה: *הפעולה מקבלת מספר שלם x* //

טענת יציאה: *הפעולה מחזירה* ***true*** *אם המספר x נמצא במחסניות הנוכחית, ו-* ***false*** *אחרת* //

<u>שאלות על מחסנית גנרית</u>

שאלה 13:

כתבו מימוש מלא של המחלקה הגנרית Stack .

שאלה 14:

ממשו פעולה המקבלת מחסנית של מחרוזות ומחזירה מחסנית שבה מופיעים אורכי המחרוזות לפי הסדר שלהן במחסנית המקורית. שימו לב: המחסנית המקורית אינה משתנה.

דוגמה: אם הפעולה מקבלת מחסנית:

"Maradona"
"Pele"
"Messy"
"Ronaldinho"

הפעולה תחזיר מחסנית ובה הערכים הבאים:

8
4
5
10

שאלה 15:

א. ממשו את הפעולה המתוארת להלן:

```
public static char memberAt(Stack<Character> st, int k)
```

טענת כניסה: *הפעולה מקבלת מחסנית st שערכיה תוים ומספר שלם חיובי k* //

טענת יציאה: *הפעולה מחזירה את הערך שנמצא בעומק k במחסנית st* //

<u>הערות:</u>

1. לאחר ביצוע הפעולה, מצב המחסנית יהיה זהה למצבה לפני ביצוע הפעולה.
2. אם במחסנית פחות מ-k ערכים יוחזר התו הריק: ' '.
3. ניתן להשתמש במחסנית עזר.

לדוגמה, אם המחסנית st נראית כך:

&
@
#
$
%

אזי לאחר הזימון `memberAt(st,4)` יוחזר התו '$'.

שאלה 16

כתבו פעולה המקבלת מחסנית, ומחזירה מחסנית חדשה, שסדר ערכיה הפוך מסדר הערכים במחסנית המקורית. אין לקלקל את המחסנית המקורית במהלך הפעולה.

שאלה 17

נתון האלגוריתם הבא:

הדפס-כבינרי (num)

*{**טענת כניסה:** האלגוריתם מקבל מספר שלם}*

*{**טענת יציאה:** האלגוריתם מדפיס את המספר num בהצגה בינרית}*

התחל עם מחסנית ריקה

חזור על הפעולה הבאה כל זמן שהמספר `num` *אינו 0:*

חלק את `num` *ב-2 ודחוף למחסנית את השארית המתקבלת מהחלוקה.*

הדפס את סדרת הספרות הבינריות שבמחסנית, לפי סדר שליפתן.

ממשו את האלגוריתם כפעולה חיצונית.

שאלה 18

כתבו את הפעולה שכותרתה מתוארת להלן:

```
public static int removeMax(Stack<Integer> s)
```

טענת כניסה: *הפעולה מקבלת מחסנית s שמכילה מספרים שלמים.* //

טענת יציאה: *הפעולה מוציאה מהמחסנית את המספר הגדול ביותר ומחזירה אותו.* //

<u>הערה</u>: בסיום, שאר המספרים במחסנית וסדרם היחסי יהיו כמו לפני ביצוע הפעולה.

שאלה 19

המחלקה IntStack המופיעה בפרק זה מוגבלת ל-100 ערכים בלבד. ממשו את המחלקה Stack הגנרית מחדש כך שמספר הערכים שנוכל לדחוף לתוכה לא יהיה מוגבל, למרות שהייצוג ימשיך להיעשות באמצעות מערך.

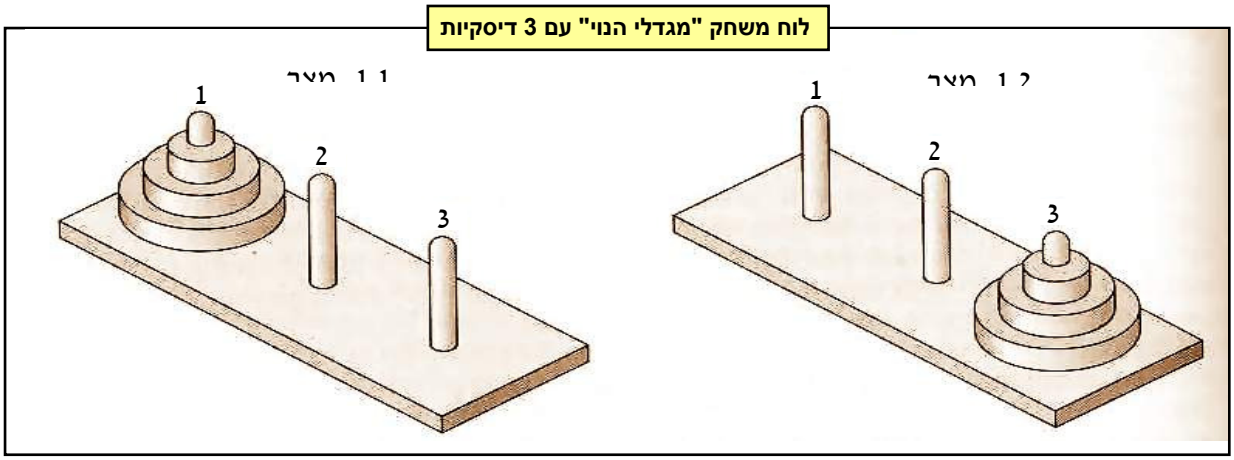
שאלה 20

<u>מגדלי הנוי - Towers of Hanoi</u>, היא חידה מתמטית שהומצאה על ידי המתמטיקאי הצרפתי אדוארד לוקאס ב-1883, והפכה למשחק חביב. המשחק בנוי מלוח שעליו נעוצים שלושה מוטות המסומנים בעזרת המספרים 1,2,3. בתחילת המשחק, על מוט מספר 1 מושחלות N דסקיות בסדר יורד ביחס להיקפן (כמתואר באיור), היוצרות צורה של מגדל - ומכאן שמו של המשחק. מטרת המשחק היא להעביר את מגדל הדסקיות בשלמותו ממוט 1 למוט 3 על פי הכללים הבאים:

א. ניתן להעביר דיסק אחד בלבד בכל שלב.

ב. באף שלב אסור שדיסק גדול יהיה מונח על דיסק קטן ממנו.

לוח משחק "מגדלי הנוי" עם 3 דיסקיות



מה עליכם לעשות?

עליכם לכתוב מחלקה בשם HanoiTowers שמגדירה את לוח המשחק "מגדלי הנוי".

המחלקה HanoiTowers צריכה לממש את הממשק הבא, המגדיר אילו פעולות פומביות יהיו למחלקה:

הפעולה בונה לוח מגדלי הנוי ובו 3 מוטות כאשר על מוט מספר 1 מושחלות numOfDiscs דסקיות בסדר יורד ביחס להיקפן – הדיסק העליון היקפו 1 ס"מ, מתחתיו דיסק נוסף שהיקפו 2 ... והדיסק התחתון ביותר היקפו numOfDiscs. שני המוטות האחרים 2,3 ריקים. (כמתואר באיור "מצב התחלתי") יש לוודא ש-numOfDiscs יהיה בין 3 ל-10 כולל. במידה ולא - יש לקבוע אותו ל-3 כברירת מחדל	`HanoiTowers(int numOfDiscs)`
הפעולה שולפת את הדיסק העליון ממוט מספר fromPoleNum ומשחילה אותו על מוט מספר toPoleNum. הפעולה מחזירה **true** אם העברת הדיסק התבצעה	`boolean moveDisc(int fromPoleNum, int toPoleNum)`

בצורה תקינה (דיסק גדול לא יהיה על דיסק קטן, והמוט fromPoleNum לא ריק), אחרת תחזיר הפעולה **false**. **הנחה:** קיימים מוטות שמספרם fromPoleNum ו-toPoleNum	
הפעולה מחזירה את מספר הדסקיות שעל לוח המשחק	`int getNumOfDiscs()`
הפעולה מחזירה את מספר הדסקיות המושחלות על מוט מספר poleNum. **הנחה:** קיים מוט שמספרו poleNum	`int getNumOfDiscs(int poleNum)`
הפעולה מחזירה את גודל הדיסק שנמצא בראש המוט שמספרו poleNum. אם המוט poleNum ריק, יוחזר הערך 0 **הנחה:** • קיים מוט שמספרו poleNum	`int getSizeTopDisc(int poleNum)`
הפעולה מחזירה **true** אם המוט שמספרו poleNum ללא דסקיות, ו-**false** אחרת	`boolean isEmpty(int poleNum)`

הערות חשובות למהלך העבודה

1. כתבו את המחלקה HanoiTowers, המגדירה לוח "מגדלי הנוי", על פי הממשק לעיל. את שלושת המוטות יש לייצג בעזרת 3 מחסניות, כאשר ערכי המחסניות יהיו מספרים שלמים המייצגים את גודל הדסקיות המושחלות על המוטות.
2. כדי להריץ את הפרוייקט כולו השתמשו במחלקה: HanoiTowersApplication.java הנמצאת במחשב שלכם במסלול הבא:

C:\eclipse\Unit 4\Help Files

תוכנית זו יוצרת את הסימולציה של המגדלים והיא משתמשת במחלקה שכתבתם בסעיף 1.

בהצלחה!