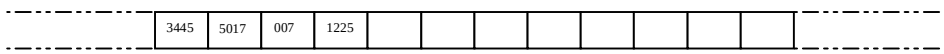
שימוש בקבצים חיצוניים

הזיכרון הפרמננטי

כל התכניות שכתבנו עד כה לא התייחסו למה שקרה לפני שהתחילו לרוץ ולא השאירו שום זכר אחרי שגמרו לרוץ. כמעט כל אפליקציה שימושית, לא מתנהגת כך. התוכנות שאנו מכירים מחיי היום יום שלנו, כמו עורכי טקסט, משחקי מחשב, דפדפנים, גיליונות אלקטרונים וכ' מסוגלות לקרוא קבצים קיימים, לכתוב לקבצים וליצור קבצים ותיקיות חדשים.
עד כה, כל התוכניות שכתבנו, השתמשו רק בזיכרון הרגיל. הזיכרון הזה הוא כידוע זמני, והאינפורמציה שבו, נשמרת רק כל עד התוכנית הספציפית רצה. אם אנו מעוניינים שהאינפורמציה תישאר גם אחרי ריצת התוכנית, עלינו להשתמש בזיכרון אחר. זיכרון זה נקרא זיכרון פרמננטי או זיכרון משני. זיכרון פרמננטי יכול להיות ממומש ע"י סוגים שונים של חומרה, לדוגמה: דיסק קשיח, טייפ, דיסקט, CD, זיכרון פלאש, שרת ברשת, מדפסת וסורק(!) ועד. בכדי שהדיון לא יהיה תלוי במדיה הספציפית בה נשמרת האינפורמציה, אנו נתייחס לזיכרון הפרמננטי באופן דומה לאופן בו התייחסנו לזיכרון הרגיל - כאל סידרה של תאים (בתים) המסוגלים לקבל ערך:



האינפורמציה הנשמרת בזיכרון הפרמננטי מנוהלת ביחידות של קבצים. בעיקרון, קובץ יכל להכיל כל מספר של בתים, גם אפס. המשמעות של הערכים המספריים הנמצאים בתאיו, נקבעת ע"י **סוג** הקובץ. סוג הקובץ נקבע בדרך כלל לפי סיומת השם שלו, המחרוזת המופיע אחרי הנקודה.
לדוגמה, הקובץ check.txt יהיה בדרך כלל קובץ טקסט, הקובץ check.jpg יהיה קובץ תמונה והקובץ check.mp3 יהיה קובץ קול. סוג הקובץ הוא מושג הדומה לטיפוס של משתנה. שניהם קובעים את האינטרפטציה של הערכים המספריים שבזיכרון. כמו ב - downcast לא אחראי, שינוי של סוג הקובץ (ע"י שינוי שמו) ללא שינוי תוכנו, יגרור אינטרפטציה שגויה של האינפורמציה.

אחד השימושים המרכזיים שעשינו בזיכרון הרגיל היה אכסון של מבני נתונים. הרצון לשימור אינפורמציה מעבר לזמן קיום התוכנית, מעלה את הצורך לשמור את מבני הנתונים גם בזיכרון בפרמננטי. למתכנת יש חופש להחליט כיצד לקודד את מבני הנתונים בזיכרון הפרמננטי. כל קידוד שיבחר, חייב להיות חד-חד ערכי בכדי שיהיה ניתן לפענוח. בנוסף, כדאי שהקידוד יבחר כך, שיהיה קל לפענוח.

עבודה בסיסית עם קבצים ב - Java

בכדי לקרוא קובץ קיים אפשר להשתמש באובייקט מסוג FileReader המהווה חלק משרותי הקלט\פלט של הספרייה הסטנדרטית. הקוד הבא מדגים קריאה פשוטה והדפסה של קובץ קיים בשם check.txt הנמצא בתיקייה הנוכחית:

```
import java.io.*;
…
void readFile() throws Exception {
        FileReader fr = new FileReader("check.txt");
        while(fr.ready())
           System.out.print((char)fr.read());
        fr.close();
}
```

ה - ctor של FileReader מקבל מחרוזת המכילה את שם הקובץ ופותח אותו לקריאה. הפונקציה ready() מחזירה ערך בולאני המציין אם הקובץ ניתן לקריאה. כאשר הקריאה מגיעה לסוף הקובץ, ready() מחזירה false.
הפונקציה read() מחזירה int שצריך להמיר בכדי להתייחס אליו כתו.

יצירת קובץ חדש וכתיבה לתוכו, יכולה להתבצע באופן דומה בעזרת אובייקט מסוג FileWriter:

```
static void writeFile() throws Exception {
        FileWriter fw = new FileWriter("check.txt");
        fw.write("One Ring to rule them all, One Ring to find them,\nOne Ring to bring
them all and in the darkness bind them");
        fw.close();
    }
```

הוספת טקסט לסוף קובץ קיים:

```
static void appendFile() throws Exception {
        FileWriter fw = new FileWriter("check.txt",true);
        fw.write("\nIn the Land of Mordor where the Shadows lie.");
        fw.close();
    }
```

קריאה וכתיבה יעילים יותר באמצעות buffering

ניתן ליעל את תהליך הקריאה ע"י שימוש במנגנון buffering - קריאה של פיסות אינפורמציה גדולות יותר לתוך מבנה של תור. הדבר נעשה ע"י שימוש במעטפת ל - FileReader הנקראת, BufferedReader. המחלקה העוטפת מספקת את אותו ממשק שמספקת FileReader אך המימוש שלה יעיל יותר. במקום לפנות לזיכרון הפרמננטי עבור כל byte בנפרד, היא ניגשת אליו רק כאשר התור מתרוקן ואז היא קוראת פיסת אינפורמציה הממלאת אותו.

```
static void bufferedReadFile() throws Exception {
        FileReader fr = new FileReader("check.txt");
        BufferedReader br = new BufferedReader(fr);
        while(br.ready())
           System.out.print((char)br.read());
        br.close();
    }
```

גם כתיבה לקובץ אפשר ליעל ע"י שימוש ב - buffering:

```
static void bufferedWriteFile() throws Exception {
        BufferedWriter br = new BufferedWriter(new FileWriter("check.txt"));
        br.write("Nine for Mortal Men doomed to die,\nOne for the Dark Lord on his
dark throne");
        br.close();
 }
```

תבנית העיצוב של מעטפות הממשות את אותו ממשק של האובייקט אותו הן יורשות, נקראת **Decorator**. תבנית זו משמשת כאשר מעוניינים להוסיף התנהגות חדשה לאובייקט, בלי לשנות את אוסף השירותים שהוא מציע. לתבנית עיצוב זו יתרון על ירושה כאשר מעוניינים בשלובים שונים של אותן התנהגויות חדשות. לדוגמה, המחלקה BufferedReader יכולה לעטוף כל Reader לאו דווקא את FileReader (שיורש מ - Reader). המחלקה BufferedReader עצמה יורשת - Reader ויש עוד מעטפות נוספות ל - Reader היורשות ממנו. עיצוב זה מאפשר "לקשט" כל אובייקט מסוג Reader בהרבה עטיפות המוסיפות לו פונקציונליות.

קריאה בשורות
ניתן גם לקרוא קובץ, שורה אחרי שורה. לדוגמה:

```
static void readFileByLine() throws Exception {
        BufferedReader br = new BufferedReader(new FileReader("check.txt"));
        String str;
        while((str = br.readLine())!= null)
           System.out.println(str);
        br.close();
 }
```

יצירת תיקייה חדשה:
יצירת תיקייה חדשה נעשית בעזרת אובייקט מסוג File והפונקציה mkdir(). המחלקה File מייצגת בעצם מסלול לקובץ (הכולל את שמו) ולכן אולי היה מתאים יותר לקרוא לה FilePath.
הנה דוגמה לשימוש:

```
static void createDir() {
        File f = new File("tmpDir");  // or File("../ex3/tmpDir") or File("/cs/stud/ori/tmpDir")
        if(!f.mkdir())
           System.err.println("couldn't create new directory");
 }
```

מסלול יחסי ומסלול אבסולוטי
כל התייחסות לקובץ ניתן לבצע ע"י שימוש במסלול מוחלט או במסלול אבסולוטי. מסלול אבסולוטי הוא מסלול המתחיל משורש עץ מערכת הקבצים. מסלול כזה מתחיל תמיד בתו '/' המסמל את אותו שורש. לדוגמה:

```
FileReader fr = new FileReader("/cs/stud/ori/check.txt");
```

פקודה זו פותחת קובץ אליו ניתן להגיע אם מתחילים משורש עץ הקבצים, נכנסים לתיקיה cs ממשיכים לתיקיה stud, משם ל - ori ומשם לתקיה ex3, שם הוא ממוקם.
היתרון של מסלול אבסולוטי הוא שהוא נשאר תקף גם אם ההרצה מתבצעת ממקום אחר בעץ הקבצים.

מסלול יחסי הוא מסלול המתחיל מהתיקייה הנוכחית, כלומר מאותה תיקיה ממנה הריצו את קובץ ההרצה. לדוגמה:

```
FileReader fr = new FileReader("../../ex3/check.txt");
```

פקודה זו פותחת קובץ הנמצא בתיקיה ex3 הנמצאת בתיקיית הסב (האב של האב) של התיקייה הנוכחית. הסימון '..' משמעו תיקיית אב.
היתרון של מסלול יחסי הוא שהוא נשאר תקף גם אם משתנים גם מיקום ההרצה וגם הקובץ אליו היא מתייחסת. זאת בהנחה שהיחס ביניהם נשמר.

במערכת ההפעלה חלונות, הכתיב של המסלול האבסולוטי הוא קצת שונה:

```
FileReader fr = new FileReader("C:/cs/stud/ori/oop2/ex3/check.txt");
```

כתיבה וקריאה של עץ מקובץ

כמעט כל תוכנית מחשב מבצעת שמירה וטעינה של מבני נתונים. הדוגמה הבאה מראה איך ניתן לשמור עץ בינארי בקובץ ואיך ניתן לטעון אותו חזרה:

```
import java.io.*;

class Tree{
   static private class Node {
         int _data;
         Node _ls, _rs;
         Node(int data) {_data = data; _ls = _rs = null; }
   }
   Node _root=null;
   public void insert(int data) {
         if(_root==null) {
            _root = new Node(data);
            return;
         }
         Node p = _root;
         while(true) {
            if(data < p._data)
                  if (p._ls == null) {
                     p._ls = new Node(data);
                     return;
                  } else
                     p = p._ls;
            else
                  if (p._rs == null) {
                     p._rs = new Node(data);
                     return;
                  } else
                     p = p._rs;
         }
   }
   public String toString() {
         return toString(_root);
   }

   private String toString(Node p) {
         if(p == null)
            return "n";
         return "("+p._data+","+ toString(p._ls)+ ","+toString(p._rs)+")";
   }

   public void save(FileWriter fw) throws Exception {
         fw.write(toString());
   }

   public void load(FileReader fr) throws Exception {
         char c = (char)fr.read();
         if(c=='n') {
            _root = null;
            return;
         }
         _root = new Node(0);
         load(fr,_root);
   }

   private void load(FileReader fr, Node p) throws Exception {
```

```
        // precondition: a tree right ahead with a missing '('
        // precondition: p points to a Node.
        // postcondition: cleans the whole tree
        p._data = readInt(fr);
        char c = (char)fr.read();
        if(c!='n') {
            p._ls = new Node(0);
            load(fr,p._ls);
        }
        fr.read(); // ","
        c = (char)fr.read();
        if(c!='n') {
            p._rs = new Node(0);
            load(fr,p._rs);
        }
        fr.read(); // ")"
    }

    private int readInt(FileReader fr) throws Exception {
        // precondition: int right ahead and ',' after it.
        // postcondition: cleans the int and the ','
        String s = "";
        char c;
        while(true) {
            c = (char)fr.read();
            if(c == ',')
                break;
            s+=c;
        }
        return Integer.valueOf(s).intValue();
    }
    static private Tree createTree() throws Exception {
        int[] array = {2,6,1,5,4,7,3};
        Tree t = new Tree();
        for(int i=0; i<array.length; ++i)
            t.insert(array[i]);
        return t;
    }
    static void checkWrite() throws Exception {
        Tree t= createTree();
        FileWriter fw = new FileWriter("t.tree");
        t.save(fw);
        fw.close();
    }
    static void checkRead() throws Exception {
        Tree t = new Tree();
        FileReader fr = new FileReader("t.tree");
        t.load(fr);
        System.out.println(t);
        fr.close();
    }
    public static void main(String[] args) throws Exception {
        //checkWrite();
        checkRead();
    }
}
```