קלט ופלט בקבצים בשפת ג'אווה - זרמים

לפניכם מדריך ג'אווה תמציתי של ביצוע פעולות קלט ופלט על קבצים באמצעות זרמים.

מעבר להכרות ראשונית עם הקונספט, הדגש הוא על הקניית כלים חיוניים לביצוע מטלות קלט ופלט בשיעורי הבית.

ניתן לקרוא ביתר הרחבה על הנושאים המוזכרים כאן, ועל נושאים יותר מתקדמים בקישור הבא:

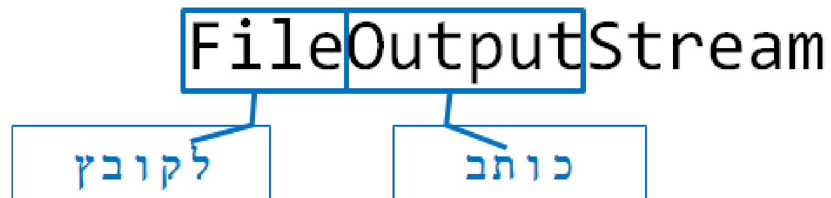
https://docs.oracle.com/javase/tutorial/essential/io/

זרמים (Streams)

קבוצה של טיפוסים שיודעים לקרוא ולכתוב ממשאבים בצורה סדרתית

- קוראים \ כותבים בתים (bytes)
- הזרימה היא תמיד חד-כיוונית
- Input Streams – לקריאה
- Output Streams – לכתיבה

לדוגמה



שימוש אופייני בזרמי קלט ופלט

כל הזרמים נפתחים עם יצירתם.

- FileOutputStream – אפילו יוצר קובץ חדש
- פתיחת זרם יכולה לגרום לזריקת חריג (Exception).

יש לסגור את הזרמים בגמר השימוש כדי לאפשר שחרור משאבים.

שימוש סטנדרטי (פסאודו קוד):

כתיבת נתונים לזרם פלט

```
Open output stream
While has data to write
      write unit

Close stream
```

קריאת נתונים מזרם קלט

```
Open input stream
While can read
      read unit
      do something
Close stream
```

דוגמאות לזרמים שימושיים (רשימה חלקית מאד)

שימוש	זרמים
קריאה\כתיבה של **בתים** (Bytes) **מקבצים**	FileInputStream, FileOutputStream
קריאה\כתיבה של **בתים** תוך שימוש ב**מאגר מובנה**	BufferedInputStream BufferedOutputStream
קריאה\כתיבה של **תווים** מ**קבצים**:	FileReader, FileWriter
קריאה\כתיבה של **טיפוסים פרימיטיביים ומחרוזות** (בדומה ל-Scanner):	DataInputStream DataOutputStream

מערכות הפעלה

כיוון שפעולות קלט ופלט ככלל קשורות לגורם חיצוני לתוכנית, לסביבה בה התוכנית רצה יש השפעה ישירה על הקוד, ובמיוחד למערכת הפעלה. בהקשר הזה, יש מספר מלכודות נפוצות, הנובעות משינויים בין מערכות הפעלה, אשר כדאי להכיר. אנו נתרכז בהבדלים בין windows למערכות הפעלה מבוססות unix (כמו לינוקס), ונדון בשוני בין התווים לירידת שורה, כמו גם בין תווי ההפרדה במסלולי קובץ.

במסגרת הקורס, חשוב במיוחד להיות מודע להבדלים הללו בהגשת שיעורי הבית. למשל, קוד שנכתב ונבדק ע"י הסטודנט על windows לא בהכרח יעבוד באותן הבדיקות בדיוק על linux.

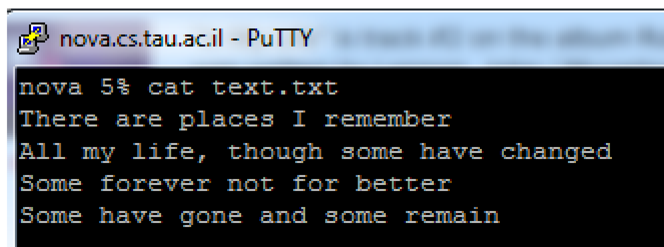
ירידת שורה

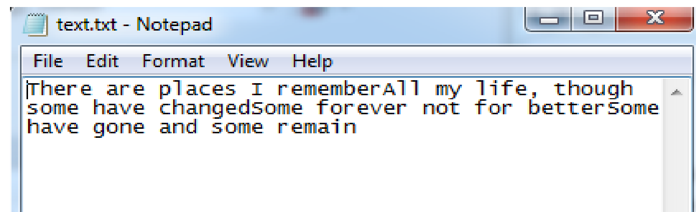
במערכות הפעלה שונות נעשה שימוש בתווי בקרה שונים עבור ירידת שורה (newline):

ב-UNIX/Linux – \n (Line Feed)

ב-Windows - \r\n (Carriage Return + Line Feed)

לפניכם דוגמה לקובץ אשר נוצר בלינוקס עם תו ירידת השורה המתאים, המוצג בשתי מערכות ההפעלה המוזכרות.





על מנת לקבל את תווי הבקרה לירידת שורה במערכת הפעלה עליה רצה התוכנית אותה אתם כותבים, יש מספר אפשרויות:

1. במתודה System.out.format, המקבילה למתודה System.out.print, שמותאמת לעריכה המחרוזת, ניתן להשתמש בn%, אשר מוחלף בתווי ירידת השורה של מערכת ההפעלה הנוכחית.

```
System.out.format ("FirstLine%nSecondLine");
```

באופן אנלוגי, ניתן גם ב-String להשתמש במתודה String.format, באותו האופן, כאשר ברצוננו לייצר אובייקט מחרוזת, במקום ישירות להדפיס אותו.

2. ניתן להשתמש במתודה `System.lineSeparator()` אשר מחזירה מחרוזת המכילה את התווים לירידת שורה.

<u>תו הפרדה</u>

כאשר אנו ניגשים לקובץ example.txt בתיקיה Software1, אם הקובץ נמצא על מערכת לינוקס תו ההפרדה יהיה / (forward slash), ואילו על windows זה \ (backslash).

לדוגמה, ליצירת מחרוזת המייצגת נתיב לקובץ בלינוקס נשתמש ב:

"Software1**/**example.txt"

ואילו, ב-windows:

"Software1****example.txt"

חשוב להיות מודעים להבדל.

ואם נרצה לרשום תו שיתאים לכל מערכת הפעלה עליה רצה התוכנית, פתרון פשוט הוא להשתמש ב File.separator:

"Software1" + File.*separator* + "example.txt"

<u>איתור התיקיה הנוכחית</u>

כאשר אנו משתמשים בכתובת רלטיבית (ובשיעורי הבית תהיו חייבים להשתמש ברלטיביות, כי הכתובת האבסולוטית שונה במחשבכם מאשר במחשבי הבודקים), נקודת הייחוס היא ביחס ל-current working directory (זאת כתובת הבסיס אליה מצורף המשך הנתיב המסופק בכתובת הרלטיבית). אם אתם לא בטוחים מהי התיקיה הנוכחית (וזה מקור שגיאות נפוץ, כאשר מקבלים הודעת שגיאה שהקובץ לא קיים, במיוחד כאשר עוברים לעבוד מה-command line), ניתן להשתמש בהדפסת עזר עם הפקודה System.getProperty("user.dir") :

```
System.out.println("Working Directory  +  System.getProperty("user.dir"));
```

אם אתם מבצעים את הבדיקה הזאת, מקמו את ההדפסה באותו חלק בתוכנית בו בהמשך תופיע פעולת הקריאה/כתיבה (או לפחות הרכבת מחרוזת הנתיב), ואל תשכחו למחוק את ההדפסה לפני ההגשה!

<u>קריאת/כתיבת קבצי טקסט</u>

בעוד שבאמצעות זרמי בתים כמו FileInputStream או FileOutputStream ניתן לעבוד ישירות עם בתים, כאשר מדובר בקבצי טקסט, פענוח וקידוד הבתים לתווים הוא תהליך מסורבל, ונרצה שכבה נוספת של אבסטרקציה על מנת לעבוד ביתר נוחות עם תווים ומחרוזות.

מחלקות בסיסיות שתואמות למשימה הזאת הן FileReader ו-FileWriter.

לפניכם, תוכנית פשוטה אשר מטרתה "להמיר" קובץ מפורמט של לינוקס לפורמט שמותאם להצגה בווינדוס, על החלפת התווים לירידת שורה. אנו מניחים כי נתיב לקובץ הקלט מסופק בארגומנט הראשון לתוכנית.

```
public class CharacterUnixToWindows {

	public static void main(String[] args) throws IOException {

		File fromFile = new File(args[0]);

		FileReader fReader = new FileReader(fromFile);

		File toFile = new File(args[1]);

		FileWriter fWriter = new FileWriter(toFile);

		char[] charRead = new char[1000];

		int numRead;

		while ((numRead = fReader.read(charRead)) != -1) {

			String string = new String(charRead, 0, numRead);

			String windowsString = string.replaceAll("\n", "\r\n");

			fWriter.write(windowsString);

		}

		fReader.close();

		fWriter.close();

	}

}
```

ראשית שימו לב ליצירת אובייקט File המייצג קובץ, ע"י קריאה לבנאי שלו עם הנתיב לקובץ מועבר כארגומנט:

```
File fromFile = new File(args[0]);
```

יצירת אובייקט זה נחוצה כארגומנט עבור הבנאי של הזרם קלט/פלט אשר מותאם לקובץ המדובר:

```
FileReader fReader = new FileReader(fromFile);
```

הפרוצדורה עבור זרם הכתיבה מקבילה לחלוטין.

המתודה fReader.read()) מקבלת כארגומנט מערך של char אליו היא תרשום את התוצאה של הקריאה, והמתודה עצמה מחזירה int אשר מציין את מספר התווים שנקראו. בכל קריאה המתודה תנסה לקרוא מספר תווים כגודל המערך, אך ייתכן שפחות תווים יקראו (אם הגענו לסוף הקובץ, או בגלל שגיאה בתהליך הקריאה). נאמר והמערך הוא בגודל 100, ונקראו רק 70 תווים, זה אומר ש-70 התאים הראשונים במערך נדרסו ע"י התווים החדשים שנקראו, אך 30 התאים האחרונים נותרו כפי שהם. על מנת לדעת אילו תווים אכן נקראו בקריאה האחרונה, חשוב לבדוק את ערך ההחזר.

כיצד אנו יודעים שהגענו לסוף הקובץ? כל זרם קלט, מחזיר סימן מיוחד ממתודת הקריאה שלו לציון סוף הקובץ. במקרה של FileReader המתודה read מחזירה -1.

שימו לב, לשורה

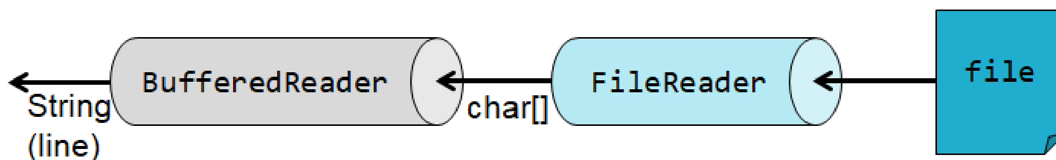
```
while ((numRead = fReader.read(charRead)) != -1)
```

מדובר בתבנית שכיחה ושימושית בעבודה עם זרמים, בה אנו בשורה אחת מבצעים איטרציה של קריאה ומוודאים לבצע את גוף הלולאה רק אם טרם הגענו לסוף הקובץ.

זרמים עוטפים (Stream Wrappers)

קיימים זרמים אשר "עוטפים" זרמים אחרים ומוסיפים להם פונקציונליות. לדוגמא, אם רוצים לקרוא מקובץ (FileReader) אבל שורה בכל פעם (BufferedReader). כשניצור את הקורא השני, נעביר לו את הראשון כארגומנט.

```
new BufferedReader(new FileReader(file))
```



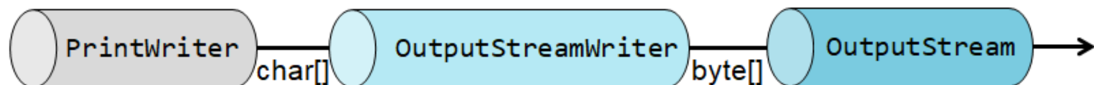
איך זה עובד?

- אנחנו נעבוד עם הזרם העוטף החיצוני ביותר (BufferedReader בדוגמא). נשלח לו מהקוד בקשות קריאה או כתיבה.
- כל זרם עוטף מחליט מתי לשלוח בקשת קריאה\כתיבה לזרם הנעטף על-ידו ומבצע עיבוד על המידע לפני שהוא מעביר אותו הלאה.
- עלינו רק לדאוג לחבר את הזרמים בצורה נכונה.
- כאשר אנו סוגרים את הזרם העוטף החיצוני ביותר, הזרמים הפנימיים נסגרים גם כן אוטומטית.

נציג דוגמה נוספת לזרם עוטף, והפעם עבור פלט ועם פונקציונליות אחרת: הזרם העוטף PrintWriter, המאפשר הדפסה בדומה ל- System.out.

```
new PrintWriter(new OutputStreamWriter(givenOutputStream))
```

תומך במתודות printf ו-format



אנו נתמקד בזרמים העוטפים BufferedReader ו-BufferedWriter, היות והם השימושיים ביותר לצרכינו בקורס. אלה הם זרמים עם מאגר מובנה (buffer) שמנוהל אוטומטית. BufferedReader יכול לקרוא תווים מעבר למה שנתבקש בקריאה, ולאחסן את התווים הנוספים במאגר המובנה, כך שבקריאות הבאות, במקום שוב לגשת לדיסק, הוא יוכל לשלוף אותם מהמאגר המובנה.

מדוע זה עדיף? מפני שבקלט ופלט פעולות גישה לדיסק הן היקרות ביותר ומהוות לרוב את צוואר הבקבוק בתוכנית. הרבה גישות לדיסק עבור קריאה של מעט תווים עולות משמעותית יותר מקריאה אחת של כל התווים בבת אחת.

המקרה של פלט הוא אנלוגי לחלוטין. הרבה כתיבות "קטנות" הן יקרות, ויהיה עדיף משמעותית, "לאגד" כמה כתיבות קטנות לפעולת כתיבה אחת עם גישה יחידה לדיסק. בהתאם, BufferedWriter "מאחורי הקלעים" לא בהכרח ניגשת לדיסק על כל פעולת כתיבה, אלא כותבת למאגר המובנה, וכאשר הוא מתמלא מבצעת כתיבה לדיסק.

עבור קבצים גדולים עבודה עם buffer היא חיונית. עבור קבצים קטנים במיוחד (מספר שורות) עדיף לעבוד ישירות עם fileReader/Writer.

נציג בפניכם כעת שכתוב של התוכנית אשר ממירה קבצים עם סיומת unix ל-windows, הפעם עם זרמים עוטפים.

```
public class BufferedUnixToWindows {

    public static void main(String[] args) throws IOException {

        File fromFile = new File(args[0]);

        BufferedReader bufferedReader =
```

```
            new BufferedReader(new FileReader(fromFile));
        File toFile = new File(args[1]);
        BufferedWriter bufferedWriter =
            new BufferedWriter(new FileWriter(toFile));
        String line;
        while ((line = bufferedReader.readLine()) != null) {
            bufferedWriter.write(line + "\r\n");
        }
        bufferedReader.close();
        bufferedWriter.close();
    }
}
```

שימו לב למספר הבדלים חשובים. ישנם מספר יתרונות בולטים בנוחות השימוש, מעבר לשיפור בביצועים. ראשית, כעת ניתן לעבוד ישירות עם מחרוזות (בלי מעבר מסורבל ממערך char) ולקרוא שורה שלמה (עד שנתקלים בתו '\n' או '\r') בפקודה אחת – readline(). מתודה זו מחזירה מחרוזת המכילה את השורה שנקראה, ואם הגענו לסוף הקובץ מוחזר הערך null (ולא 1- כמו בתוכנית הקודמת). יחד עם זאת, עדיין ניתן להשתמש גם במתודה read.

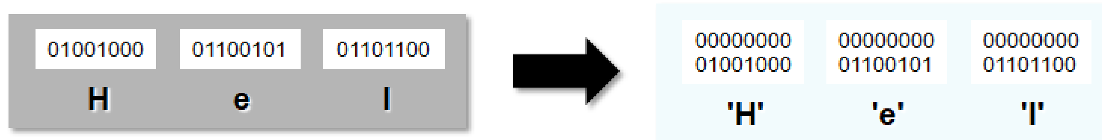
הדוגמה האחרונה תשמש אותכם כתבנית בסיסית (לא כולל הפעולה הקונקרטית של החלפת תווי ירידת השורה) עבור קלט/פלט עם קבצים באמצעות זרמים.

קידוד

בעיה פוטנציאלית: Characters בג'אווה הם עם קידוד מסוים (UTF-16). מה אם לקבצים שלנו יש קידוד אחר?

והפתרון...

בד"כ Java פותרת את הבעיה בעצמה! קידוד ברירת מחדל מוגדר עבור מערכת ההפעלה. Java מתרגמת אותו ל-characters שלה.



חשוב להיות מודעים לכך שאכן מתרחש קידוד שאינו ברור מאליו "מאחורי הקלעים", אך אנו בקורס לא נתעסק בבעיה זו, ונניח שמשימת הקידוד נפתרת אוטומטית.

יחד עם זאת, אם יש צורך לציין מפורשות את הקידוד, ניתן לעשות זאת באמצעות המחלקה InputStreamReader באופן הבא:

```
File fromFile = new File(args[0]);

FileInputStream fis = new FileInputStream(fromFile);

InputStreamReader ISR = new InputStreamReader(is, Charset.forName("UTF-8"));

BufferedReader BR = new BufferedReader(ISR);
```

בדוגמה זו הכיתוב שהוגדר הוא UTF-8. תחילה יצרנו זרם מטיפוס FileInputStream אשר עובד ישירות עם בתים. לאחר מכן, עטפנו אותו עם הזרם InputStreamReader אשר ניתן להגדיר בבנאי שלו ארגומנט שני המייצג את הקידוד. ולבסוף, על מנת לשוב לכלי העבודה המוכרים לנו, עטפנו אותו ב-BufferedReader , דרכו נוכל לבצע את הקריאה בהמשך.

מה עדיף – Scanner או FileReader יחד עם BufferedReader?

ל Scanner יש Buffer, אבל הוא קטן יותר מה Buffer של ה-BufferedReader. גודל ה-Buffer הוא רלוונטי כאשר נעבוד עם קבצים גדולים ונרצה לחסוך גישות לדיסק (עדיף 10 קריאות של 4096 בתים מאשר 4096 קריאות של 10 בתים (המספרים הם שרירותיים)!

Scanner מאפשר פעולות עיבוד מתוחכמות על קובץ הטקסט אותו אנו קוראים, ומפרק את הקלט לטוקנים. בעוד שה-BufferedReader מחזיר מחרוזות בלבד, ה-Scanner יכול לחלץ טיפוסים פרימיטיביים כמו boolean, int וכו'. שימושי כאשר אנחנו רוצים לבצע המרות תוך כדי הקריאה מהקובץ.