קלט ופלט בקבצים בשפת ג'אווה - זרמים

לפניכם מדריך ג'אווה תמציתי של ביצוע פעולות קלט ופלט באמצעות Scanner וזרמים.

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

ניתן לקרוא ביתר הרחבה על הנושאים המוזכרים כאן ועל נושאים יותר מתקדמים בקישורים הבאים:

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

https://docs.oracle.com/javase/8/docs/api/java/util/Scanner.html

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

זרמי קלט ופלט הם קבוצה של מחלקות שבאמצעותן נלמד כיצד לקרוא ולכתוב לקבצים, כאשר כל אחת מותאמת יותר לתרחיש אחר. המחלקה Scanner מהווה ביחס אליהן מין "אולר שוויצרי" עבור קלט של טקסט. השימוש בה נעשה על ידי פעולות שהן יותר high-level ונוחות לשימוש ככלל, אך לפעמים פחות מותאמות למקרים מסוימים. עם Scanner נלמד מעבר לקריאה מקבצים, גם קבלת קלט מהמקלדת וקריאת מילים מתוך מחרוזות.

הנושאים שהמדריך מכסה לפי הסדר הם:

- פיצול מחרוזת לתת-מחרוזות (הכנה לדיון על Scanner).
- Scanner.
- כתיבת קוד שעובד על פני מערכות הפעלה שונות
  - ירידת שורה בקבצים
  - תו הפרדה בין תיקיות בנתיב
- נתיבים אבסולוטיים ורלטיביים
- איך לאתר את נקודת היחוס בכתובת רלטיבית (ואיך לדבג את תרגילי הבית בלי לאבד שפיות).
- זרמי בתים.
- זרמי טקסט.
- דיון על קידוד (חדשות טובות: לא נתעסק בקידוד בקורס)
- השוואה בין Scanner לזרמים.
- המחלקה StringBuilder בה עליכם להשתמש כאשר אתם מבצעים ברצף שינויים רבים על אותה מחרוזת, בשביל למנוע בעיות יעילות זמן וזיכרון.

טרמינולוגיה של Split

לפני שנבחן את המחלקה Scanner, נדון תחילה בטרמינולוגיה מקובלת לפעולת ה-split, אשר מפרקת מחרוזת למערך של תת-מחרוזות. המתודה split של המחלקה String מקבלת פרמטר שנקרא delimiter, שהינו סדרת תווים אשר מפרידה בין כל זוג תת-מחרוזות עוקבות. התת-מחרוזות אשר נמצאות במערך הפלט נקראות tokens.

לפניכם דוגמה בסיסית של שימוש ב-split עם delimiter שהינו תו רווח בודד " ".

```
String str = "A string with a single space delimiter";
String[] tokens = str.split(" ");
System.out.println(Arrays.toString(tokens));
```

הפלט הינו:

```
[A, string, with, a, single, space, delimiter]
```

ה-delimiter למעשה אינו מוכרח להיות מחרוזת "פשוטה". זה רק מקרה פרטי של מנגנון מורכב יותר שנקרא ביטוי רגולרי, עליו תלמדו בשלב יותר מתקדם של התואר, ואין כל צורך להתמצא בו בקורס זה. בשפה פשוטה מדובר ב"נוסחה" שיכולה לתפוס במכה אחת קבוצה רחבה של מחרוזות שונות עם דפוס משותף מסוים. הסינטקס למי שלא מאומן בביטויים רגולריים נראה קריפטי.

השימוש הנפוץ ביותר ב-split הוא אכן לפצל משפט למערך של מילים, כאשר ה-delimiter הוא רווח לבן. אך במקרים רבים לא מובטח שמדובר ברווח יחיד. ההפרדה בתוך טקסט (אם נתעלם מכל עניין הפיסוק) היא רצף whitespace שכולל כמות מסוימת של רווחים, ירידות שורה וטאבים. ניתן לתפוס את כל הקומבינציות תחת ההגדרה של whitespace עם הביטוי הרגולרי "\\s+", כפי שנראה בדוגמת הקוד הבאה. שימו לב, שאם נרצה להימנע גם מהרווחים בקצוות של המחרוזת, פתרון אחד הוא להשתמש במתודה ()trim:

```
String str = " A   string with   \n  various whitespace delimiters";
String[] tokens = str.split(" ");
System.out.println(Arrays.toString(tokens));

tokens = str.split("\\s+");
System.out.println(Arrays.toString(tokens));

tokens = str.trim().split("\\s+");
System.out.println(Arrays.toString(tokens));
```

הפלט הינו:

```
[, A, , , string, with, , ,
, , various, whitespace, delimiters]
[, A, string, with, various, whitespace, delimiters]
[A, string, with, various, whitespace, delimiters]
```

Scanner

המחלקה java.util.Scanner היא סורק טקסט פשוט. אובייקט Scanner מקבל בבנאי מקור טקסט כלשהו, כאשר שלושת הדוגמאות שנלמד עליהן במדריך זה הן: מחרוזת, קובץ או המקלדת. הסקנר "עוטף" (ראו גם דיון על זרמים עוטפים בהמשך) את מקור הטקסט, ומבצע עליו פעולת split. אך הוא אינו מבצע את ה-split בבת-אחת, אלא באופן הדרגתי בהתאם ל"פקודות next" שאנו מריצים, אשר בכל פעם מחזירות את הטוקן הבא. מעבר לכך, שנוח באופן זה לקבל, למשל בלולאה, בכל איטרציה את המילה (או המשפט) הבאה, יש לסקנר גם מתודות קריאה מיוחדות, אשר מבצעות אוטומטית המרה לטיפוס פרימיטיבי. למשל, אם המילה הבאה הולכת להיות מספר שלם, אז במקום לקרוא עם הפקודה ()next אשר מחזירה מחרוזת, ניתן להשתמש בפקודה ()nextInt אשר מחזירה int. יש כמובן גם מתודות nextDouble וכו'.

כברירת מחדל ה-delimiter של Scanner מוגדר להיות כל רצף whitespace, אך ניתן לשנות זאת דרך המתודות שלו.

לפניכם דוגמה בסיסית לשימוש בסקנר אשר מאותחל עם מחרוזת בתור מקור הקלט. הערך המוחזר מופיע בהערה לצד כל פקודת קריאה).

```
Scanner scanner = new Scanner("12 12.4 the long\nand winding road ...");
int anInt = scanner.nextInt(); // 12
float aFloat = scanner.nextFloat(); // 12.4
String aString = scanner.next(); // the
String aLine = scanner.nextLine(); // long
String bLine = scanner.nextLine(); // and winding road ...
```

- Scanner, כמו גם הזרמים שתלמדו עליהם בהמשך, נפתח ונסגר. חישבו עליו בתור ברז. הפתיחה נעשית אוטומטית ביצירה שלו. ואילו את הסגירה עליכם לעשות ידנית עם פעולה close() בתום השימוש. **חשוב מאד לבצע את הסגירה בתום השימוש**, כדי, בין היתר, להימנע מ"דליפת משאבים" (דיון זה יורחב במסגרת הקורס "מערכות הפעלה").

- מה יקרה אם נבצע קריאת next כאשר הגענו לסוף הטקסט, ואין מילה נוספת להחזיר? תיזרק שגיאת זמן ריצה. במקרים רבים, אנחנו לא יודעים האם כבר הגענו לסוף, במיוחד כאשר אנו משתמשים בלולאה, ולכן מקובל לפני קריאת next() לקרוא ל-hasNext() אשר מחזירה ערך בוליאני. לכל פקודת next יש מקבילת has, כמו hasNextInt() ו-hasNextLine() וכו'.

  לפניכם דוגמת קוד עם תבנית סטנדרטית לקריאה עם סקנר באמצעות לולאה:

```
String input = "1 fish 2 fish red fish blue fish ";
Scanner s = new Scanner(input).useDelimiter(" fish ");
while (s.hasNext()) {
        System.out.println(s.next());
}
s.close()
```

אשר מדפיס:

```
1
2
red
blue
```

- כאשר נרצה לקרוא קלט אשר המשתמשת מזינה דרך המקלדת, אנו נאתחל את הסקנר עם האובייקט System.in אשר מייצג את מקור הקלט הסטנדרטי – המקלדת. האובייקט הזה הוא מופע ספציפי שכבר מאותחל עם תחילת התוכנית, ואין כל צורך לבצע בעצמכם קריאה כלשהי לבנאי.
  ה-system.in הוא סוג של זרם (עליהם נדבר בהמשך המדריך) וגם הוא יכול להיסגר. כאשר אנחנו סוגרים סקנר, הוא סוגר גם את מקור הטקסט שלו. זה תקף לא רק לקבצים אלא גם ל-System.in. וכאן יש לזכור נקודה מאד חשובה – את ה-System.in ניתן לסגור רק פעם אחת במהלך ריצה של תוכנית ג'אווה, ואז לא ניתן לפתוח אותו מחדש. לכן, **סגירה של**

סקנר שעוטף את System.in צריכה להיעשות רק לאחר שאין יותר שום פונקציונליות בהמשך הריצה שצפויה לעשות שימוש ב-System.in.

לאחר אתחול הסקנר, הפונקציונליות למעשה כמעט זהה למקרה בו הוא אותחל עם מחרוזת או קובץ, אך במקרה של קלט אינטראקטיבי עם המקלדת יש כמה נקודות שוני שכדאי להכיר.

ראשית, כאשר הסקנר יבקש לקרוא את המילה הבאה והמשתמשת תזין את הטקסט, לחיצה על רווח לא שולחת סיגנל חיובי ל-hasNext או ל-next שמודיע שהתקבלה עוד מילה. זה קורה רק בלחיצה על Enter. כלומר, הסקנר יקבל את הטקסט במשפטים ולא במילים. מבחינת התוצאה הסופית אין שום הבדל. למשל, אם יש לולאה שמבקשת בכל פעם לקרוא את המילה הבאה, והמשתמשת רשמה חמש מילים ואחריהן לחצה על Enter, אז מיד אחרי אותה לחיצה יתבצעו חמש איטרציות רצופות של הלולאה.

נקודה חשובה נוספת היא עצירת הקלט. כיצד המתודה hasNext (אם בכלל משתמשים בה במקרה של קלט מהמקלדת - במקרים רבים אנו יודעים מראש בדיוק כמה מילים או משפטים צריכים להיכתב, ואז אין לנו את הבעיה הזאת) יודעת להבדיל בין מקרה בו המשתמשת "לוקחת את הזמן" לפני (או תוך כדי) שהיא מקלידה, לבין מקרה שבו היא לא מתכוונת להקליד יותר? איך אי פעם יתקבל false? התשובה היא שהמתודה לא יודעת להבדיל, וכדי שהיא תחזיר false המשתמשת צריכה להזין את התו המיוחד EOF (end of file). אופן ההזנה שלו משתנה בין מערכות הפעלה (וניתן לבדוק אותו בגוגל), אך לרוב מדובר ב ctrl+D או ctrl+Z.

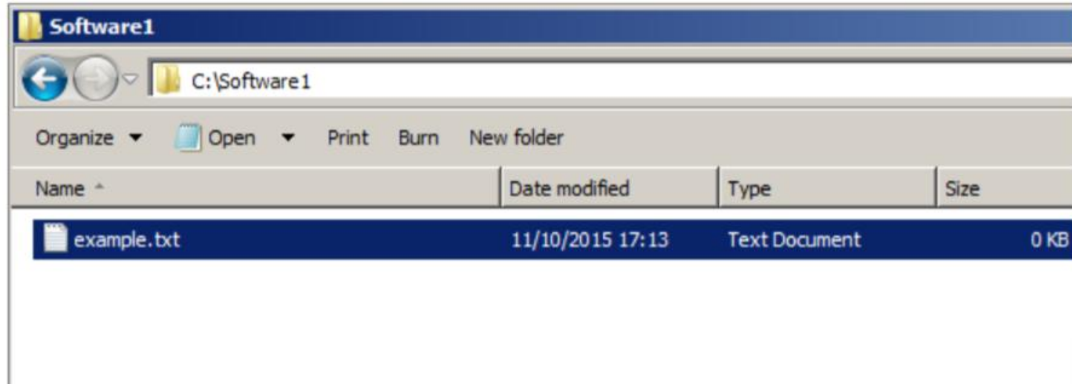
לפניכם דוגמת קוד אשר מקבלת בלולאה קלט מהמקלדת. מומלץ לנסות להריץ את הקוד הזה באקליפס, ולפתח אינטואיציה לאופן השימוש:

```
Scanner s = new Scanner(System.in);
System.out.println("enter line:");
while (s.hasNext())
 System.out.println(s.next());
s.close();
```

- כאשר נרצה לקרוא טקסט מתוך קובץ, אנו נאתחל את הסורק עם מופע של המחלקה java.io.File. מופע זה יקבל בבנאי שלו מחרוזת שהיא הנתיב של הקובץ אליו הוא מתייחס. כאשר אנו מעבירים לסקנר קובץ, מיד מתקבלת שגיאת קומפילציה. זאת שגיאה טיפוסית אשר מופיע במחלקות שונות כאשר מבצעים קריאה מקובץ. הסיבה לכך היא שבפתיחת קובץ עלול להיזרק חריג (Exception), למשל אם הקובץ לא קיים או נעול, ויש מנגנון שכופה על המתכנתת להתייחס למקרה הזה.
ההתייחסות יכולה להיות קוד מסוים שצריך לרוץ במקרה של שגיאה (כמו להדפיס הודעה למשתמש) או לשים הצהרה (כמו תווית אזהרה) ליד חתימת המתודה בה אנו כרגע נמצאים אשר אומרת שעלול להיזרק חריג.
אנו נלמד על חריגים בשלב מתקדם יותר של הקורס, ונבין למה בחלק מהחריגים יש את המנגנון הזה, שכופה את ההתייחסות של המתכנתת, וכיצד ניתן לכתוב קוד (בבלוק של try-catch) אשר רץ במקרה בו נזרק החריג. אך, בשלב זה אנו פשוט נדע שזה משהו שקיים כאשר קוראים מקובץ, ונוסיף למתודה בה אנו נמצאים סעיף שמתחיל במילה throws ואחריו

השם של החריג שעלול להיזרק, או חריג כללי יותר. Exception עצמו הוא חריג כללי שתמיד אפשר לרשום במקום החריג היותר ספציפי במקרה הזה שהוא FileNotFoundException (שתי האפשרויות שקולות בשלב זה מבחינתנו). החדשות הטובות הן שהאקליפס כהרגלו (לאחר לחיצת עכבר מתאימה) מציע לנו פתרון לשגיאת קומפילציה, ואפשר לבחור שם באפשרות של להוסיף הצהרת throws, כך שלא נצטרך להקליד כלום בעצמינו.

נניח שאני רוצה לקרוא את הקובץ הבא במחשב שלי:



לפניכם קוד שעושה זאת, ומדפיס בלולאה את כל המילים שהוא קורא:

```
public static void main(String[] args) throws FileNotFoundException {
	String pathname = "C:\\Software1\\example.txt";
	File f = new File(pathname);
	Scanner s = new Scanner(f);
	while (s.hasNext()) {
		String word = s.next();
		System.out.println(word);
	}
	s.close();
}
```

כמובן שאת האתחול ניתן לבצע גם בשורה אחת, למשל:

```
Scanner s = new Scanner(new File("C:\\Software1\\example.txt"));
```

נציין כבר עכשיו שבמקום לרשום מפורשות את הנתיב לקובץ בתוך הקוד, מומלץ להעביר אותו בתור ארגומנט לתוכנית, כדי לחסוך הרבה כאב ראש.

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 מודעים להבדלים הללו בהגשת שיעורי הבית. למשל, קוד שנכתב ונבדק ע"י הסטודנטית על windows לא בהכרח יעבוד באותן הבדיקות בדיוק על linux.

בכל מקרה, נלמד כאן כיצד לכתוב קוד כללי יותר, שהוא אגנוסטי למערכת ההפעלה שעליה הוא ירוץ.

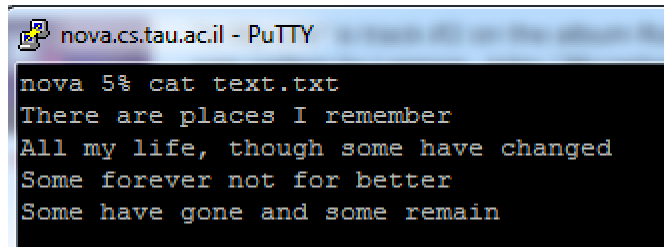
ירידת שורה

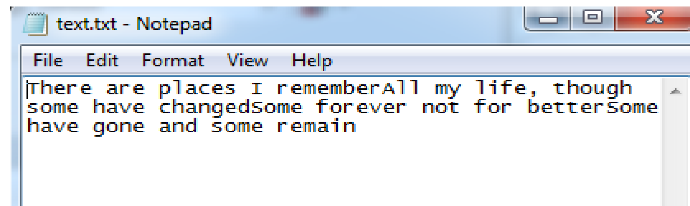
במערכות הפעלה שונות נעשה שימוש בתווי בקרה שונים עבור ירידת שורה (newline):

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 (בתרגילי הבית החוסר אחידות של ירידת השורה עקרונית רק לכתיבה לקבצים, ולא להדפסות למסך).





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 ב%n, 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.

תו הפרדה

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>נתיב אבסולוטי לעומת רלטיבי</u>

הקונספט של נתיב אבסולוטי לעומת רלטיבי הינו חשוב במיוחד להכיר, גם מעבר לג'אווה או אפילו תכנות (למשל, גם בעבודה עם command line). נתיב אבסולוטי במחשב מתייחס ל"כתובת המלאה" של קובץ או תיקיה. כלומר הוא "מתחיל בהתחלה". למשל, ב-windows קל לזהות שהנתיב `C:\\Software1\\example.txt` הינו אבסולוטי, שכן הוא מתחיל באות של כונן ואחריה נקודתיים ושני סלאשים. במקרה של לינוקס קל לזהות נתיב אבסולוטי אם התו הראשון בו הוא סלאש קדמי.

בכתובת רלטיבית, לעומת זאת, נקודת ההתחלה היא לא ה"התחלה של המחשב" אלא "כאן" (לרוב נקרא current directory או working directory), והנתיב הוא ביחס לנקודת ההתחלה הזאת. בכל רגע נתון, בין אנחנו בסייר הקבצים במחשב, או ב-command line או בתוכנית ג'אווה, יש פרמטר (לפעמים סמוי) שמציין באיזו תיקיה אנחנו נמצאים כרגע.

אם, למשל, אנחנו עובדים ב-command line של לינוקס, ואנחנו נמצאים בתיקיה z שהנתיב האבסולוטי שלה הוא /x/y/z, ואנחנו רוצים לעבור לתיקיה w שנמצאת תחתיה, אז הרבה יותר נוח לרשום:

cd w

מאשר

cd /x/y/z/w

במקרה הראשון השתמשנו בכתובת רלטיבית. שימו לב, כי הסינטקס של כתובות רלטיביות מאפשר גם "לעלות למעלה" (כלומר "לחזור אחורה") במבנה התיקיות, ע"י 2 נקודות. למשל:

cd ../../w

יעביר אותנו לתיקיה w אחרת (אם היא קיימת) שהנתיב המלא שלה הוא /x/w ("עלינו" פעמיים למעלה עד x ואז "ירדנו" ל-w).

באיזו סוג נתיב מומלץ להשתמש?

לפני שנענה על כך, נדגיש שבאופן כללי, מומלץ מאד שהנתיב אשר התוכנית תשתמש בו יועבר אליה בתור ארגומנט. באופן, זה אם תחליטו לשנות או לתקן נתיב של קובץ, או להעביר בכל פעם קובץ אחר, או להריץ את הקוד במחשבים שונים (למשל אחד מהם שלכם, אחד שרת האוניברסיטה, ואחד של הבודק), במקום כל פעם לשנות את הנתיב שרשום בקוד, הרבה יותר נכון פשוט לשנות כל פעם את הקריאה לתוכנית שהקוד שלה נותר זהה, וכל השוני מתבטא בנתיב שהיא מקבלת בתור

ארגומנט. שגיאה מאד נפוצה היא שהקוד רץ היטב על המחשב של הסטודנט, ואז מתקבל ציון אפס, כי הקוד מניח שלבודק יש את אותו קובץ הקלט ובאותו המיקום במחשב.

במקרים כאלה מומלץ, אך כלל לא חובה, שהארגומנט יהיה כתובת רלטיבית, כי נוח יותר לעבוד עם כתובות כאלו, וכדאי להתרגל אליהן. אם ממש מסתבכים ניתן לעבור לאבסולוטית.

לעומת זאת, אם אנחנו מקודדים את הכתובת לתוך הקוד עצמו, אז ברוב המקרים – ותמיד במקרה של תרגילי הבית! – חובה להשתמש בכתובת רלטיבית, כי ההתחלה של הכתובת האבסולוטית תשתנה בין מחשב למחשב. באקליפס, למשל, "נקודת ההתחלה" של כתובת רלטיבית נקראית ה-project root. אם, למשל, מדובר בתיקיה src, אז כל מבנה התיקיות תחתיה נשמר גם כאשר src מועתקת למחשב של הבודקים. ולכן, הנתיב הרלטיבי של כל תיקיה תחת src הוא זהה בכל המחשבים אליהם src הועתקה.

<u>איתור התיקיה הנוכחית</u>

החדשות הרעות, בעיקר כאשר נתקלים בכך בפעם הראשונה, היא שזה לא ברור מאליו להבין מה התיקיה הנוכחית בקונטקסט של תוכנית ג'אווה. וגרוע מכך, זה יכול להשתנות בין הרצה במחשב שלנו באקליפס לבין הרצה ללא אקליפס בשרת, שם פתאום הכתובת הרלטיבית שהעברנו כארגומנט כבר לא מתאימה, למרות שמבנה הפרויקט נותר כפי שהוא.

בשלב זה סטודנטים רבים מתייאשים, ומחליטים לפרוש מהתואר או להפוך למתרגלים. אין כל צורך להרוס לעצמכם את החיים, או לפרוש מהתואר.

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 הנתיב מועבר כארגומנט אז אחרי שנבין איך צריך לשנות את אותו (לרוב להוסיף עליה או ירידה של תיקיה אחת בתחילתו), אחרי נסיון או שניים נעביר את הארגומנט הנכון.

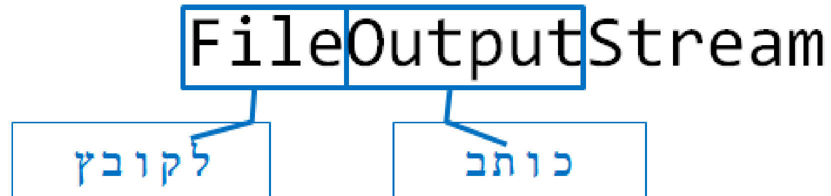
אם אתם מבצעים את הבדיקה הזאת, מקמו את ההדפסה באותו חלק בתוכנית בו מיד בהמשך תופיע פעולת הקריאה/כתיבה (או לפחות הרכבת מחרוזת הנתיב). תשתדלו כמובן שהפקודה תופיע בטסטר שלכם, אותו אנחנו בכל מקרה לא בודקים, כי אנחנו משתמשים בטסטרים משלנו (אחרת - אל תשכחו למחוק את ההדפסה לפני ההגשה!). שימו לב, שברוב המקרים לא צריך לדאוג שזאת בעיה שתחלחל לבדיקה של התרגיל שלכם, כי הבודקים יודעים להעביר נתיבים נכונים. הבעיה תצוץ בעיקר כאשר אתם בודקים את הקוד של עצמכם.

<u>זרמים (Streams)</u>

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:



<u>שימוש אופייני בזרמי קלט ופלט</u>

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
```

<u>דוגמאות לזרמים שימושיים (רשימה חלקית)</u>

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**	DataInputStream DataOutputStream

(בדומה ל-Scanner):	

<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

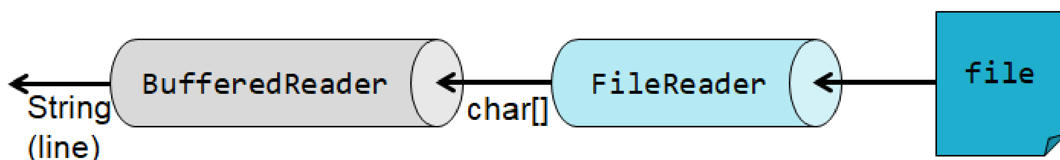
```
while ((numRead = fReader.read(charRead)) != -1)
```

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.

<u>זרמים עוטפים (Stream Wrappers)</u>

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))
```



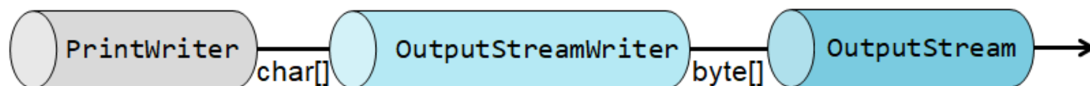
איך זה עובד?

- 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.

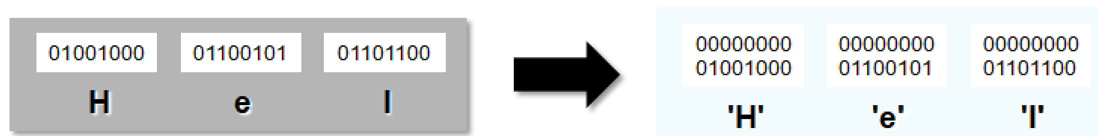
הדוגמה האחרונה תשמש אותכם כתבנית בסיסית (לא כולל הפעולה הקונקרטית של החלפת תווי ירידת השורה) עבור קלט/פלט עם קבצים באמצעות זרמים.

קידוד

בעיה פוטנציאלית: Characters בג'אווה הם עם קידוד מסויים (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(fis, Charset.forName("UTF-8"));

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

בדוגמה זו הקידוד 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.

<u>מה עדיף – Scanner או FileReader יחד עם BufferedReader?</u>

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'. זה שימושי כאשר אנחנו רוצים לבצע המרות תוך כדי הקריאה מהקובץ.

<u>המחלקה StringBuilder</u>

למדנו בקורס שמחרוזות הן immutable. לא ניתן לשנות אובייקט String קיים, אלא רק ליצור מופע חדש עם השינוי הרצוי. אם אנחנו מבצעים בלולאה הרבה שינויים על המחרוזת, כמו לצרף בכל פעם לסוף שלה מחרוזת נוספת, מדובר לפיכך במימוש מאד לא יעיל (כל מחרוזות הביניים קיימות לרגע קצר ואז נזרקות כאשר נוצרת בכל פעם מחרוזת חדשה אליה צריך להעתיק "מהתחלה" את כל התווים), ולפעמים לפגיעה מורגשת בביצועים.

במקרים כאלה, במקום להשתמש ב-String, מומלץ להשתמש במחלקה StringBuilder. מחלקה זו גם יכולה לייצג מחרוזות, אך היא mutable, ולכן ניתן לבצע במחרוזת שינויים בלי ליצור אובייקט חדש בכל פעם. במקום אופרטור השרשור + נשתמש במתודה append. ניתן להמיר בכל שלב למחרוזת על ידי המתודה ()toString.

קיימות כמובן מתודות נוספות, ובמידת הצורך הינכם מוזמנים לעיין בתיעוד של המחלקה:

https://docs.oracle.com/javase/8/docs/api/java/lang/StringBuilder.html

לפניכם קוד בסיסי אשר מוסיף למחרוזת abc את התו d, ומדפיס את התוצאה המרשימה :

```
StringBuilder sb = new StringBuilder("abc");
sb.append("d");
System.out.println(sb);
```