תרגיל 2

תכנות מונחה עצמים

הנה שוב הקישור להנחיות ההגשה: דף ההנחיות

.

הנה קבצי הבדיקה לכל שאלה:

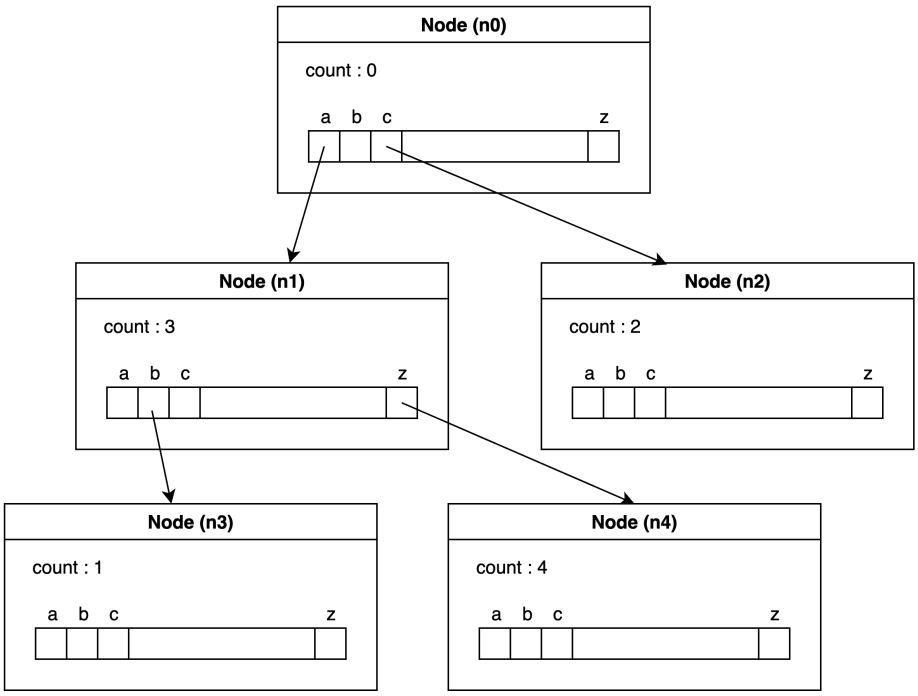
- שאלה 1, בחבילה tree, הקובץ TreeTester.java.
- שאלה 2, בחבילה bank, הקובץ BankTester.java
- שאלה 3, בחבילה tasks, הקובץ TasksTester.java.
- שאלה 4, בחבילה game, הקובץ GameTester.java.

מהנחיות תרגיל 1, אולי הכי חשובות הן:

1. לא לשנות את הקבצים שאתם מורידים, ואם אתם משנים אז הורידו מחדש.
2. לראות שאין אף סימן אדום על אף קובץ (לא שלכם ולא שלי), כי זה אומר ששאלה זו לא תתקמפל (או כל השאלות אם זה למשל Tester.java)
3. ליצור את קובץ הזיפ להגשה אך ורק בעזרת הרצת Tester.java.

שאלה 1 (package tree)

שאלה זו לא משתמשת כלל בירושה. נבנה בה עץ לשמירת מחרוזות המורכבות מאותיות קטנות באנגלית. קודקוד של עץ יהיה מופע של המחלקה Node, כאשר לכל קודקוד כזה 26 מצביעים לילדים, כשאולי חלק מהמצביעים הם null. בנוסף יהיה לו גם מספר המתאר את מספר הפעמים שהמחרוזת הספציפית מופיעה בעץ. כך למשל:



השם בסוגריים הוא רק כדי שיהיה לנו נוח לדבר על הקודקודים השונים. העץ הזה מתאר מצב בו הכניסו לתוכו:
שלוש פעמים את "a", פעמיים את "c", פעם אחת את "ab", וארבע פעמים את "az".
במחלקה ישנם השדות והשיטות הבאות (כרגיל, שדות ושיטות פרטיות אתם יכולים להוסיף כרצונכם):

private int count	מספר הפעמים שהמחרוזת המתוארת ע"י המסלול מהשורש עד לקודקוד זה הוכנסה לעץ.
private Node[] children	מערך המצביעים לילדים. אורכו הוא 'z' - 'a' + 1.
public int num(String s)	מחזירה את מספר הפעמים שהמחרוזת s נמצאת בעץ, כאשר חושבים על הקודקוד הנוכחי בתור השורש.
public void add(String s)	מוסיפה את המחרוזת s לעץ, כאשר הקודקוד

	הנוכחי נחשב השורש.

הנה מה שתחזיר כל אחת מהקריאות הבאות:

n0.num(“c”) → 2
n0.num(“az”) → 4
n1.num(“b”) → 1
n1.num(“”) → 3
n0.num(“cc”) → 0

הוספת מחרוזת יכולה כמובן לגרום להוספת קודקודים חדשים. למשל,

n0.add(“cc”)

תוסיף עוד קודקוד חדש עם count = 1, ללא ילדים (כל המערך יהיה null), והמערך של n2 יצביע עליו באינדקס c במערך.

הקוד הבא (גם בכל סדר אחר של שורות), יוצר את העץ בתמונה (בלי שמות הקודקודים), כאשר המשתנה root מצביע לשורש:

```
Node root= new Node()
root.add(“c”);
root.add(“c”);
root.add(“ab”);
root.add(“az”); root.add(“az”); root.add(“az”); root.add(“az”);
root.add(“a”); root.add(“a”); root.add(“a”);
```

ועכשיו אפשר לשאול למשל root.num("ab"), ולקבל 1.

הערות:

1. כמובן שאין דבר כזה מערכים שהאינדקס שלהם הוא אותיות. אבל, char הוא מספר לכל דבר, ולכן למשל אפשר לכתוב `a` - `z`, ולהשתמש בזה כדי להמיר אות לאינדקס, וגם להיפך.
2. אתם יכולים גם להוסיף בנאי ללא פרמטרים אם אתם רוצים, אבל לא חייבים (אפשר פשוט לאתחל את השדות כבר בזמן ההגדרה שלהם).
3. העץ הזה בנוי טיפה אחרת ממה שאתם רגילים: אין שום מידע בקודקודים עצמם (חוץ מ-count). המידע הוא בעצם במצביעים - במסלול שמייצג את המילה. בפרט, **אין צורך** להוסיף שום שדה נוסף לקודקודים.
4. ברקורסיה תצטרכו לקחת את המחרוזת שקיבלתם s ולייצר מחרוזת חדשה שהיא בדיוק כמו s רק בלי האות הראשונה. לשם כך תוכלו להשתמש בשיטה String.subString, חפשו עליה קצת והבינו איך להשתמש בה.
   - ברקורסיה, כדאי לחשוב על מקרה הבסיס כמקרה בו המחרוזת s היא ריקה: "", ולא על המקרה בו אורך המחרוזת הוא 1.

5. למעשה, המימוש הזה לא יעיל בגלל הקריאה ל-substring, כי היא מייצרת מחרוזת חדשה כל פעם. אפשר לפתור את הבעיה הזאת, אבל זה מסבך קצת. אז ויתרנו :)

כדי להדגים שימוש במחלקה זו (וזה כמובן חלק מדרישות התרגיל), כתבו מחלקה בשם ReversedWords, שיש לה שיטה אחת עם החתימה:

```
public static int checkReversed()
```

שיטה זו קוראת מהמשתמש (הכוונה ל- System.in) מחרוזת אחרי מחרוזת ע"י שימוש ב-Scanner, ותעצור כאשר תקרא את המחרוזת "X". היא מחזירה את מספר המחרוזות שכבר הופיעו קודם בקלט בהיפוך סדר אותיות (היפוך מדויק, לא סתם ערבוב). כמובן, עליכם להשתמש במחלקה Node כדי לשמור את המחרוזות שכבר נראו.

למשל, אם הקלט הוא:

evil stressed star live raw pupils slipup where raw war rats live madam X

אז נחזיר 5 כי כל מילה באדום נספרת. שימו לב ש:

- war נספרת רק פעם אחת למרות שיש שתי מחרוזות raw לפניה.
- madam לא נספרת כלל, למרות שהיא היפוך של עצמה. כמובן שאם היתה מופיעה עוד madam אז היא כן היתה נספרת.

אתם יכולים להניח שהקלט הוא חוקי ומורכב אך ורק ממילים שמורכבות מאותיות קטנות באנגלית (מלבד ה-X בסוף).

שאלה 2 (package bank)

זוהי שאלה פשוטה המתרגלת שימוש בירושה. כתבו קודם את המחלקה Account המתארת חשבון בנק פשוט:

public Account(String name)	בנאי המאתחל את החשבון עם השם הנתון, ועם 0 שקלים בחשבון.
public int getShekels()	מחזירה את מספר השקלים בחשבון.
public String getName()	מחזירה את שם החשבון.
public void add(int amount)	מוסיפה amount שקלים לחשבון. שימו לב ש-amount יכול להיות מספר שלילי (ואז יורד מהחשבון כסף).
public String toString()	מחזירה את שם החשבון וכמה שקלים יש בו כרגע. למשל: Pinoccio has 150 shekels

לאחר מכן, כתבו את המחלקה ProAccount שהיא תת מחלקה של Account, ובה כל חשבון זוכר את כל ההיסטוריה שלו. כלומר כמה שקלים היו בו לאחר כל פעולה. יש למחלקה זו השיטות הבאות, כשחלקן למעשה דורסות את השיטות של Account:

public ProAccount(String name)	בנאי זהה לזה של Account (זכרו שבנאים לא מקבלים בירושה, לכן חייבים לכתוב אותם).
public void add(int amount)	כמו ב-Account, אבל זוכרת את כמות השקלים לאחר הפעולה.
public String toString()	מחזירה את שם החשבון וכמה שקלים יש בו, ובנוסף את היסטורית החשבון (ראו למטה).
public static void transfer(ProAccount from, ProAccount to, int amount)	שיטה סטטית המעבירה amount שקלים מחשבון from אל חשבון to (למה שיטה זו היא סטטית?)

אם למשל נריץ את הקוד הבא,

```
ProAccount a = new ProAccount("Shimshon");
ProAccount b = new ProAccount("Yovav");
a.add(1000);
```

```
ProAccount.transfer(a, b, 100);
a.add(200);
ProAccount.transfer(a, b, 50);

System.out.println(a);
System.out.println(b);
```

נקבל את ההדפסה:

```
Shimshon has 1050 shekels [1000,900,1100,1050]
Yovav has 150 shekels [100,150]
```

הערה:
ניתן להניח שההיסטוריה של חשבון מסוג ProAccount היא לכל היותר בגודל 100. כלומר השיטה add נקראת לכל היותר 100 פעמים בחשבון כזה.

שאלה 3 (package tasks)

בשאלה זו נבנה שתי מחלקות לסידור אוטומטי של משימות, כאשר יש מגבלות סידור עליהן. המחלקה הראשונה תקרא Tasks. במחלקה זו כל משימה מיוצגת ע"י מספר בלבד, והמשימות שלנו הן מ-0 ועד num-1. אלו השיטות שלה:

public Tasks(int num)	תאתחל את המערכת לשימוש עם num משימות.
public boolean dependsOn(int task1, int task2)	תוסיף למערכת את התלות האומרת ש-task1 לא יכולה להתבצע לפני task2. אם task1 או task2 אינם מספרים חוקיים של משימות היא תחזיר false ואחרת true.
public int[] order()	תחזיר מערך בו מופיעות כל המשימות לפי סדר שמקיים את כל התלויות שקיבלנו. אם אין כזה סדר (כלומר יש מעגל של תלויות), אז היא תחזיר null.

למשל:

```
Tasks t = new Tasks(6);
t.dependsOn(3, 2);
t.dependsOn(0, 3);
t.dependsOn(2, 5);
t.dependsOn(4, 5);
System.out.println(Arrays.toString(t.order()));
```

יכולה להדפיס (ויש הרבה אפשרויות נכונות):

[1, 5, 2, 3, 0, 4]

לעומת זאת, אם הייתה גם השורה:

```
t.dependsOn(5, 3);
```

אז אמור היה להיות מודפס null.

הערות:

1. אתם יכולים להניח שהקלט הוא חוקי.
2. ניתן להניח שאחרי הקריאה ל-order אף אחד לא ישתמש יותר במופע הזה של המחלקה.
3. אין דרישה לאלגוריתם יעיל פה, אבל אתם מוזמנים לעשות אותו כמה יעיל שבא לכם, זה יכול להיות מאתגר ומעניין.
4. מומלץ לנסות ולפתור את הבעיה האלגוריתמית לבד (היא לא מאוד קשה), לשם כך הנה שתי המלצות מאוד כלליות:
   - ציירו לעצמם דוגמאות כגרף מכוון ונסו לחשוב מה לעשות קודם בלי קוד בכלל. לאחר מכן הבינו איך לעשות זאת בקוד - למשל הסתכלו על איזשהו גרף לדוגמא, ונסו להבין איך אפשר לאתר איזושהי משימה שיכולה להיות מבוצעת ראשונה..
   - השאירו את עניין זיהוי המעגלים לסוף. רוב הסיכויים שהוא ייפתר כמעט מעצמו אם תפתרו את המקרים שאין בהם מעגלים.

מי שבכל זאת רוצה הנחיות, הנה רעיון האלגוריתם (**התרעת ספוילרים!**):

- שמרו את כל התלויות כמערך דו מימדי, או בכל דרך אחרת.
- עבדו בשלבים, ובכל שלב מצאו משימה שלא תלויה באף משימה אחרת. הוסיפו אותה למערך התוצאה, ומחקו את התלויות שהיא משתתפת בהן. אם לא מצאתם כזאת, כנראה שיש מעגל.
- אפשר לעשות את המחיקות לא ממש של התלויות, אלא ע"י מונים - אבל לא חובה.

שלב ב:

כתבו מחלקה NamedTasks, שיורשת מהמחלקה Tasks, ובה כל משימה מתוארת ע"י שם ולא מספר.
יהיו לה השיטות הבאות:

בנאי המקבל מערך של שמות המשימות	public NamedTasks(String[] names)
תוסיף למערכת את התלות האומרת ש-task1 לא יכולה להתבצע לפני task2. אם task1 או task2 אינם שמות של משימות קיימות היא תחזיר false, ואחרת true.	public boolean dependsOn(String task1, String task2)
תחזיר מערך בו מופיעות כל המשימות לפי סדר שמקיים את כל התלויות שקיבלנו. אם אין כזה סדר (כלומר יש מעגל של תלויות), אז היא תחזיר null.	public String[] nameOrder()

כלומר, אלו שיטות בדיוק כמו אלו של המחלקה Tasks, רק שהן עובדות עם מחרוזות ולא מספרים.

- חשוב ביותר שלא תכתבו את האלגוריתם מחדש, אלא תשתמשו בצורה חכמה בשיטות שירשתם מ-Tasks.
- גם פה, אין צורך להקפיד על יעילות.
- וגם פה, אפשר להניח שהקלט חוקי.

למשל:

```
String[] names = {"zero", "one", "two", "three", "four", "five"};
NamedTasks t2 = new NamedTasks(names);
t2.dependsOn("three", "two");
t2.dependsOn("one", "three");
t2.dependsOn("two", "five");
t2.dependsOn("four", "five");
System.out.println(Arrays.toString(t2.nameOrder()));
```

יכולה להדפיס למשל (ושוב, יש הרבה אפשרויות):

```
[zero, five, two, three, one, four]
```

שאלה 4 (package game)

בשאלה זו נבנה מחלקה שתנהל משחק שני שחקנים פשוט, שממנו נוכל לרשת בקלות ולהגדיר שני משחקים מוכרים: איקס עיגול ו-4 בשורה.

המחלקה Player

ראשית נגדיר מחלקה פשוטה Player שמתארת שחקן בודד:

public Player(String name, char mark)	מאתחלת שחקן עם שם ועם אות אחת שתהיה הסימן שלו בלוח המשחק.
public String getName()	מחזירה את השם.
public char getMark()	מחזירה את הסימן.
public String toString()	מחזירה למשל לשחקן עם השם pilpel וסימן P, את המחרוזת: pilpel(P)

המחלקה Board

עתה נבנה את המחלקה Board שתחזיק לוח משחק:

protected Player[][] board	לוח המשחק עצמו, בו בכל מקום מופיע השחקן שתפס את הנקודה הזאת בלוח.
protected int n,m	גודל לוח המשחק.
public Board(int n, int m)	מייצרת לוח משחק בגודל n על m.
protected boolean set(int i, int j, Player p)	אם מקום i,j ריק, אז מסמנת שהוא נלקח ע"י השחקן p ומחזירה true. אחרת מחזירה false.
public boolean isEmpty(int i, int j)	מחזירה true אם המיקום ה-i,j ריק בלוח.
public Player get(int i, int j)	מחזירה מצביע לשחקן שתפס את מקום i,j בלוח, או null אם המקום ריק.

public boolean isFull()	מחזירה אמת אם כל הלוח מלא.
public String toString()	מחזירה את לוח המשחק כמחרוזת. ראה למטה דוגמא.
protected int maxLineContaining(int i, int j)	מחזירה את אורך הקו הישר הארוך ביותר (כולל אלכסונים) בלוח המכיל את הנקודה i,j וכולו מסומן ע"י אותו שחקן כמו הנקודה i,j.

למשל:

```
Player p1 = new Player("Bibi", 'B');
Player p2 = new Player("Gantz", 'G');
Board b = new Board(3,4);
b.set(0, 0, p1);
b.set(1, 0, p1);
b.set(2, 2, p2);
b.set(0, 0, p2);
b.set(0, 1, p1);
System.out.print(b);
```

תדפיס:

```
BB..
B...
..G.
```

כלומר, toString מייצר מחרוזת עם הסימן של כל שחקן במקום ששייך לו, נקודה במקומות הריקים, ויורד שורה בסוף כל שורה (כולל השורה האחרונה).

שימו לב שכל הקריאות של set מלבד הרביעית מחזירות true.

בנוסף, הקריאות הבאות מחזירות:

```
b.maxLineContaining(1, 0) → 2
b.maxLineContaining(2, 2) → 1
```

אפשר להניח בשיטה זו שאין null במיקום ה-i,j.

הערות:

- המשתנים board ו-n,m הם לא פרטיים, וזה כדי שיהיה קל לגשת אליהם מהמחלקות שתירשנה את Board. מצד שני הם לא פומביים, כדי להגביל את הגישה אליהם.
- כך גם השיטות set ו-maxLineContaining. זאת כיון שאנחנו לא רוצים שהן תהיינה פומביות, או כדי שמשתמש מבחוץ לא ישתמש בהן ישירות (ב-set), או כי הוא לא צריך אותן (maxLineContaining).

לגבי maxLineContaining:

אתם חופשיים לכתוב איזה מימוש שאתם רוצים לשיטה זו. העיקר פה הוא לא סיבוכיות אלא בהירות והימנעות משיכפול קוד. זו שיטה יחסית מורכבת ולכן יש פה נטייה לא קטנה לבאגים. לכן, כתבו לעצמכם הרבה טסטים שבודקים כל מני מקרים: את כל הכיוונים של קווים (אלכסונים או ישרים), את זה שהנקודה היא בקצה אחד או בקצה השני או באמצע, שיש עוד קווים קצרים יותר שהנקודה משתתפת בהן, וכו'. הטסטים שנתתי לכם ממש לא בודקים הכל!

תנסו ותראו להימנע משכפול קוד זה לא כל כך פשוט פה. הכי טוב זה לנסות ולפתור את זה לבד, אבל מי שרוצה מוזמן לקרוא את ההמשך עם הצעה לפתרון יחסית אלגנטי:

נכתוב שיטת עזר שמחשבת את אורך הקרן הכי ארוכה שיוצאת מנקודה נתונה לכיוון נתון ומכילה את אותו שחקן כמו בנקודה הנתונה. יש לנו שמונה כיוונים אפשרים (ימינה, שמאלה, למטה, למעלה, אלכסון למעלה ימינה, אלכסון למעלה שמאלה, וכו'). איך נתאר כיוון? בעזרת שני מספרים dx ו-dy, שיכולים לקבל את הערך 1, 0 או 1-. למשל הכיוון למטה שמאלה יהיה dx=-1, dy=1 (כאשר אנחנו חושבים על 0,0 בתור הפינה השמאלית העליונה). הכיוון dx=0 ו-dy=1 מתארת את הכיוון "למטה". כלומר חתימת השיטה תהיה:

```
private int rayLength(int x, int y, int dx, int dy)
```

ברגע שיש לנו את השיטה הזאת, לא יהיה קשה לכתוב את maxLineContaining.

המחלקה Game

עתה נבנה מחלקת משחק ראשוני שיורשת מ-Board הנקראת Game. במשחק פשוט זה, הראשון שתופס את המיקום 0,0 מנצח.

protected Player[] players	מערך השחקנים במשחק.
protected Scanner s	יאותחל לקרוא מהקלט הסטנדרטי.
public Game(int n, int m, Player p1, Player p2)	מאתחל את הלוח לגודל הנתון וישמור את שני השחקנים.
protected boolean doesWin(int i, int j)	אם המהלך האחרון מנצח, כשהמהלך היה במיקום i,j, תחזיר true אחרת false. אפשר להניח שהקלט תקין ואין צורך לבדוק אותו.
protected boolean onePlay(Player p)	תבקש מהשחקן p להכניס את המהלך שלו, ותחזיר true אם זהו מהלך מנצח.
public Player play()	תבקש משחקן אחד לשחק, אחרי זה שחקן שני, וחוזר חלילה. זה עד אחד השחקנים מנצח, ואת היא תחזיר אותו, או שהלוח מתמלא ואי אפשר לשחק יותר, ואז תחזיר null (כרגע נראה לא

הגיוני שהלוח יתמלא, אבל במחלקות שיירשו ממחלקה זו, תנאי הנצחון ישתנה ואז זה כן יתאפשר)	

הערות:

- השיטה play תשתמש בשיטה onePlay שתשתמש בשיטה doesWin. הפירוק פה נועד לזה שכשנירש ממחלקה זו, נוכל לשנות את הכללים בקלות מבלי לשכפל קוד.
- השיטה onePlay יכולה להניח שהקלט חוקי, אבל אם מישהו מנסה לתפוס נקודה שכבר נתפסה, היא תבקש מהשחקן להכניס ערכים שוב.

למשל, משחק שנוצר כך:

```
Game g = new Game(3, 4, new Player("Red", 'R'), new Player("Black", 'B'));
g.play();
```

יכול להיראות ככה:

```
Red(R), please enter x and y: 1 1
....
.R..
....

Black(B), please enter x and y: 2 1
....
.R..
.B..

Red(R), please enter x and y: 1 1
There is a piece there already...
Red(R), please enter x and y: 0 0
R...
.R..
.B..

Red(R) Won!
```

כאשר מה שמסומן באדום הוא מה שהמשתמש מכניס.

הערות:

- אין חשיבות לטקסט שמודפס, אלא רק להתנהלות המשחק (כלומר, לא נבדוק את ההדפסות שלכם, אלא רק את מצב הלוח בכל שלב, ואת התוצאה הסופית).

המחלקה TicTacToe

אחרי כל ההכנה הזאת, יהיה מאוד קל לכתוב את המחלקה TicTacToe שיורשת מ-Game וממשת את המשחק איקס עיגול (ככה קוראים לו באנגלית). במשחק זה, השחקן הראשון מקבל את הסימון X והשחקן השני את O. כמו כן הלוח הוא תמיד בגודל 3 על 3. מנצח מי שמייצר קו באורך 3.

public TicTacToe(String player1, String player2)	מייצרת משחק בו שם השחקן הראשון הוא player1 ושם השני הוא player2.
protected boolean doesWin(int x, int y)	דורסת את שיטה זו כך שתעבוד למשחק איקס עיגול.

המחלקה FourInARow

נצטרך להתאמץ טיפה יותר כדי לכתוב את המחלקה FourInARow שיורשת מ-Game וממשת את המשחק 4 בשורה. מי שלא מכיר מוזמן לשחק כאן.

ההבדל העיקרי במשחק זה, הוא שלא ניתן לשים כלי משחק בכל מקום בלוח, אלא רק להגיד באיזו עמודה שמים אותו, והוא נופל עד למטה.

public FourInARow(String player1, String player2)	יוצרת לוח משחק בגודל 6 על 7 עם שחקנים עם שני השמות. סימן השחקן הראשון הוא 'W', והשני הוא 'B'.
protected boolean doesWin(int i, int j)	דורסת את השיטה המקורית מחזיר אמת אם כלי המשחק במיקום זה יוצר קו באורך 4.
protected boolean onePlay(Player p)	דורסת את השיטה המקורית כך שבכל פעם יתבקש מהשחקן מס העמודה בה הוא רוצה להכניס כלי משחק. אם העמודה מלאה, השחקן יתבקש לבחור עמודה שוב.

למשל, הרצת משחק בעזרת השורות הבאות:

```
FourInARow g = new FourInARow("White", "Black");
g.play();
```

יכולה להיראות כך:

```
White(W), please enter column:1
.......
.......
.......
.......
.......
.W.....

Black(B), please enter column:0
.......
.......
.......
.......
.......
BW.....

White(W), please enter column:1
.......
.......
.......
.......
.W.....
BW.....

Black(B), please enter column:2
.......
.......
.......
.......
.W.....
BWB....
```

וכך הלאה...