**מחלקת ממשק UserInterface**

**הערות:**

מינוח "הגרלת קידוד"- הכוונה שהתכנית מגרילה מחרוזת כלשהי של אותיות. לא רציתי להשתמש במינוח "ניחוש" כדי לייצר הבדלה ברורה בין התכנית שמגרילה לבין השחקן שמנחש.

מינוח "המרה לכיוון הלוגי" – הכוונה לביצוע המרה של הערכים שקיבלנו מן השחקן לערכים שהלוגיקה יודעת להתמודד איתם. במקרה שלנו זה קבלה של סטרינג ע"י הקונסול והמרתו לתווים (כי הלוגיקה שלנו משתמשת בתווים). באופן דומה היינו יכולים לקבל "כלבים וחתולים" והיינו צריכים להמיר לתווים כדי שהלוגיקה תדע להתמודד איתם.

מינוח "המרה לכיוון הממשקי" – הכוונה לביצוע המרה של ערכי הלוגיקה לערכי הממשק. הפעולה ההפוכה למעשה לפעולה הקודמת.

**הגדרות:**

Enum עבור מקשי מקלדת של כן/לא/יציאה.

**דאטא ממברס:**

המשחק הלוגי m\_Logic שאחראי על כל החישובים והלוגיקה שמתבצעת מאחורי הקלעים.

**מתודות:**

**מתודה Run-**

1. קוראת למתודה לקבלת מספר הניחושים
2. קוראת למתודה להגרלת קידוד
3. עבור i מספר הניחושים נבצע:
   1. מנקה את המסך
   2. קריאה למתודה להדפסת סטטוס משחק.
   3. קריאה למתודה לקליטת ניחוש מן השחקן לשורה i
   4. קריאה למתודה לחישוב תוצאה של הניחוש לשורה i
   5. אם (קריאה למתודה בוליאנית שבודקת אם ניצחנו בשורה i)
      1. Break ללולאה
4. (פה צריך עדיין להחליט כיצד אנחנו רוצים שהמשחק יעבוד. מפורט בהרחבה בעמוד האחרון תחת "הבהרות כלליות").

**Q - יציאה? - כרגע להבנתי לא עוצר**

**מתודה GetUserGuess לקבלת מספר הניחושים-**

1. מוצאת את הגבולות למספר הניחושים.
2. מדפיסה בקשה לשחקן להקיש מספר ניחושים בטווח נתון.
3. מקבלת מספר ניחושים בstring וממירה אותו לushort.
4. אם המספר בטווח אז נעדכן את הפרמטר של מספר הניחושים שנמצא בלוגיקת המשחק.
5. אחרת, נדפיס הודעת שגיאה ונריץ לולאה מחדש.

**מתודה CheckIfLegal בוליאנית לבדיקת תקינות הקלט-**

1. מבצעת בדיקת תקינות הקלט מבחינה ממשקית (הקלט בפורמט המתאים, כלומר רווח מפריד, ובאורך המתאים).
2. קוראת למתודה לבדיקת תקינות לוגית של הקלט. (הקלט בטווח ערכים המתאים).
3. אם הקלט עומד בתקנים הנ"ל נחזיר true.

* שניתי שם ל-isLegalInput
* הייתה שם בדיקה חסרת משמעות - legalInput=true תמיד

**מתודה GetUserGuess קליטת ניחוש מן השחקן לשורה i-**

1. הדפסת בקשה לקבלת קלט
2. קליטת הקלט של המשתמש לתוך string (זו הברירה היחידה מאחר ואנו משתמשים בממשק של קונסול(.
3. קריאה למתודה בוליאנית לבדיקת תקינות הקלט
4. אם הקלט תקין נבצע:
   1. קריאה למתודה להכנסת הקלט לטבלה בשורה i
5. אחרת:
   1. הדפסת שגיאה וביצוע לולאה מחדש.
6. אחרי שקיבלנו קלט מהשחקן וראינו כי הוא תקין צריך לקרוא למתודה ממשקית שתהיה אחראית להמרה לכיוון הלוגי ואז תקרא למתודה לוגית שתכניס את הקלט לBoard במקום ה-i.

(לא ממומש עד הסוף)

**מתודה לחישוב תוצאה של הניחוש לשורה i-**

1. קוראת למתודה לחישוב תוצאה של הניחוש בשורה ה-i. החישוב מתבצע בלוגיקה של המשחק.

(לא ממומש)

**מתודה בוליאנית שבודקת אם ניצחנו בשורה i-**

צריכה לקרוא למתודה לוגית שבודקת האם ניחשנו נכון בשורה ה-i. אם כן אז המתודה צריכה להדפיס למסך הודעה מתאימה. אבל צריך למממש את המתודה רק אחרי שנחליט כיצד מסתיימת המתודה Run ואיך ניתן להריץ אותה שוב.

(לא ממומש)

**מתודה PrintCurrentBoardStatus הדפסה סטטוס משחק-**

1. מדפיסה כותרות מתאימות
2. קוראת להדפסת לוח המשחק

**מתודה PrintBoard הדפסת לוח המשחק-**

1. עבור i מספר הניחושים נבצע:
   1. קריאה להדפסת שורה (i)

**מתודה PrintBoardLine הדפסת שורה (i) בלוח-**

1. מדפיסה נקודותיים ורווח
2. עבור כל j מספר האותיות לניחוש נבצע:
   1. נקרא לפונקציית הדפסת הנתון (i,j)
   2. נדפיס רווח
3. נדפיס נקודותיים
4. עבור v מספר הוויים נבצע:
   1. נדפיס וי
   2. רווח
5. עבור x מספר האיקסים נבצע:
   1. נדפיס איקס
   2. רווח
6. נדפיס נקודותיים ונרד שורה
7. נקרא למתודה להדפסת גבול

**מתודה PrintGuess הדפסת נתון (i,j)-**

המקום בו צריך לדאוג להמרה לכיוון הממשקי לפני שמדפיסים. כלומר לבצע המרה הפוכה מchar לפורמט של הממשק (כי הממשק לא בהכרח משתמש בchars אלא אולי ב"כלבים וחתולים"). ורק אז להדפיס את הנתון הבודד.

**מתודה PrintBorder הדפסת גבול-**

מדפיס את הגבול לפי הפורמט ולפי הגודל המתכוונן. נעזרת במתודת העזר printDuplicateChar.

**מחלקה לוגית GameLogic**

**הערות:**

**הגדרות:**

Enum עבור ייצוג הגבולות למספר הניחושים.

Enum עבור ייצוג גבולות טווח הערכים בקידוד.

Class BoardLine שמייצג שורה בטבלה: מכיל מערך בגודל GuessArraySize של אותיות ושני פרמטרים: כמות האיקסים וכמות הוויים. (הכנתי ממנו class כדי שיהיה ניתן להכין עבורו גטרים/סטרים והעמסת אופרטור).

**דאטא ממברס:**

קבוע **GuessArraySize** המחזיק את אורך הקידוד.

קבוע **NotExists** המחזיק את הערך שיסמל עבורנו אות שאינה קיימת בקידוד.

פרמטר **UserGuessesAmount** המחזיק את מספר הניחושים הרצוי.

1- יסמל אות שלא נמצאת בקידוד, וכל מספר אחר יסמל את מיקום האות ביחס לקידוד (כלומר אם יש מספר שהוא לא 1- אזי האות קיימת בקידוד). השתמשתי בשם Goal כדי שיהיה ברור לאורך הקודם שהמערך הנ"ל מחזיק את השאיפה של השחקן/המטרה/היעד. דרך זו עדיפה מהסיבות הבאות:

א) מאפשר גישה בO(1) בעזרת פונקציית ערבול. כלומר שיפרנו את יעילות התכנית.

ב) אנו נמנעים מלהחזיק בכל תא במערך הנ"ל את ייצוג האות (הרי לא נזדקק לכך אם יש לנו פונקציית ערבול) ונמנע מלהחזיק משתנה בוליאני שייתן אינדיקציה ל"האם האות קיימת או לא", כיוון שבעזרת סימול מתאים ניתן לקבל אינדיקציה זו בכל מקרה.

חשוב להבהיר כי במקרה זה היעילות עולה על הזיכרון (שיטה אחרת אשר מייצגת זיכרון על פני יעילות היא גישה בo(n) על ידי חיפוש האות המתאימה, במערך/סטרינג שמחזיק רק את האותיות שבקידוד). לכן כדי לא לפגוע בזיכרון יותר מידי נאלצנו "לקצץ" את המערך הנ"ל לכדיי נתון אחד בתא (ולא שלושה נתונים כמו שרצינו בהתחלה: אות, הימצאות, מיקום).

מערך **Board** בגודל UserGussesAmount כאשר כל תא יכיל את האיבר BoardLine. מערך זה מייצג את לוח המשחק כולו (ניחושים ותוצאות). דרך זו עדיפה מהסיבות הבאות:

א) האיבר BoardLine קודם כל מכיל בתוכו מערך שמייצג את הניחוש של השחקן. דבר זה מקל עלינו מבחינת הגישה ומונע הסתבכויות מיותרות עם string. (זה רעיון זהה למטריצה שרצינו בהתחלה, ובעזרת העמסת האופרטורים זה גם קריא).

ב) האיבר BoardLine מכיל בתוכו את הייצוג של כמה איקסים וכמה וויים יש לשורה הספציפית (כלומר התוצאה), ובכך מונע מאיתנו להחזיק תאים ריקים שהיו עשויים להיות במטריצה/מערך תוצאות. (חסכון בזיכרון).

ג) שיפור בקריאות. בעזרת האיבר ייצרנו קשר ישיר והדוק בין השורה לבין התוצאה שלה. זאת לעומת הרעיון המקורי ליצור שתי מטריצות שבהן צריך להבין כי התוצאה בשורה ה-i של מטריצת התוצאות שייכת לניחוש בשורה ה-i של מטריצת הניחושים.

**מתודות:**

**גטר/סטר של UserGuessesAmount**

**גטר/סטר של GuessArraySize**

**גטר של Board**

**העמסת אופרטור של BoardLine**

**מתודה initiateGame לאתחול המשחק כולו-**

1. מאתחל את הלוח לפי גודל הניחושים ויוצר BoardLine בכל תא במערך Board. (חשוב לשים לב שבקונסטרקטור של BoardLine הערך שמתווסף לכל תא הוא רווח).
2. אתחול ערכי היעד בערכים רנדומליים.

**מתודה initiateGameGoalValues לאתחול המערך GameGoal-**

מאתחל את כל הערכים של הGameGoal להיות NotExists.

**תקינות לוגית של הקלט-**

1. בדיקה שהקלט עומד בטווח ערכים שהקצנו ב-eGuessLetterBounds.
2. בדיקה שאין אות חוזרת על עצמה. (עוד לא ממומש)

**בדיקת ניצחון בשורה i**

בודק האם בשורה הi הצלחנו לנחש נכון. אם מספר ה-Vים שווה למספר האותיות בקידוד אז נחזיר true, אחרת false.

**המחלקהBoardLine**

**הגדרות:**

Enum עבור ייצוג תוצאה V/X

**דאטא ממברס:**

מערך UserGuess המייצג את הניחוש.

ExistsWrongPlaceResult מייצג את מספר ה-X עבור אותו ניחוש.

ExistsRightPlaceResult מייצג את מספר ה-V עבור אותו ניחוש.

(בכוונה האותיות X ו-V לא מיוצגות בשם של הדאטא ממברס כדי לשפר את הקריאות של הקוד, מה גם שאולי בעתיד נתחרט ונרצה לייצג את הערכים באותיות שהם לאו דווקא X ו-V).

הבהרות כלליות:

1. מראש לא נתתי דגש על תקינות הקוד מבחינת הסטנדרטים של גיא, אז סביר להניח שיש הרבה ליקויים בגזרה הזו.
2. בבדיקת תקינות הקלט צריך לבדוק גם שאין אות שחוזרת פעמיים. גיא אומר שאפיון המשחק הוא לשיקולנו.
3. צריך לחסוך באופן משמעותי ב-UserInterface את השימושים הנרחבים ב-Console.Write/WriteLine. אם נרצה בעתיד לשנות את ממשק המשתמש למשהו שאינו חלונאי אז נצטרך למעשה לבנות UserInterface שלם מחדש. לעומת זאת אם נצליח איכשהו לצמצם את שימושים הנ"ל לכדי יצירת סטרינגים או שימוש ב-stringBuilder אז יהיה עלינו בעתיד לשנות כמה שורות בודדות שיחליפו את Console.Write וזהו.
4. צריך להחליט כיצד אנו רוצים לממש את התכנית כאשר שחקן מגיע לסיומו של המשחק. הרי ברור כי ה-Run צריך לרוץ מחדש, אך מי קורא לו? האם בסוף ה-Run נקרא ל-Run הבא אם ברצוננו להמשיך לשחק (חסרון: הזיכרון לא ימחק עד אשר נסיים לשחק בכל המשחקים), או שנצא מ-Run ואז במידה ונרצה להמשיך לשחק נקרא לו שוב דרך Program?
5. בעיה בהדפסות – כשאני מדפיס את במתודה GetUserGuess את הערך ‘A’+i אני לא מקבל אות, אלא ערכי אסקי למסך. ואם אני מנסה להדפיס את הערך שנמצא ב-quitKey אני לא מקבל 'Q' אלא ממש את השם quitKey.
6. יש המון שכפול של קוד באזור ההדפסות.