Coding Basics

Setup

- Use the same exercise-runner project for all exercises below.
- For each exercise, open a different JavaScript file (01.js, 02.js.).
- Copy the assignment as a comment to the top of your JavaScript file.
- At the top of your JavaScript file write to the console the number and purpose of the exercise i.e. “Ex03: Convert Celsius to Fahrenheit”.
- The output to the console should be easy to understand, i.e ‘Got 2 numbers: 2, 5 - the biggest number is: 5’.

Basics

1. Read (prompt) a first name and a last name. Declare the variable *fullName*, and then welcome the user by his full name.

2. Read two numbers and print (to the console) the result of the following operations on them:
   (% ,/ ,*)

3. Read a temperature in Celsius from the user, and print it converted to Fahrenheit.

4. Read a number from the user for distance and a number for speed and print the time.

$$\text{Time} = \frac{\text{Distance}}{\text{Speed}}$$

   (Now let's build waze ;))

5. Read 3 digits from the user and print the number in full:
   for example: if the user entered the numbers 3,2,6, we should store them in a variable holding the value of 326 and then print that variable to the console.
   - **BONUS**: In this case, working with strings is easier, try solving the task while using numeric variables.

Conditions

6. Read 3 variables from the user: a, b, c. These will be the a, b, c variables of a quadratic equation. (משוואה ריבועית)

    a. Calculations for the solution of the quadratic equation:

    - Print to the console the value of "-b"
    - Print to the console the value of "2*a"
    - Print to the console the value of the **discriminant**. Discriminant=b*b-4*a*c

    b. **BONUS**:

    Now, a quadratic equation looks like:

    $$ax^2 + bx + c = 0$$

    The two solutions for of this equation are X1 and X2:

    $$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

    Your tasks:

    - Print the quadratic equation as a string to the console
    - Print the solutions of X1 and X2 to the console.

    Example: for the following equation: $2X^2 – 5x + 2 = 0$

    Your inputs are: a=2, b=-5, c=2

    your output to the console should be:

    $2X^2 – 5x + 2 = 0$

    x1 = 2 ; x2 = 0.5

    Hint: To print the x^2 to the console, use this: string: 'x\u00B2'

7. Read 3 numbers from the user and check if the 3rd is the sum of the first two, if so, print all the numbers to the console in this way: 6 + 4 = 10

8. Read 3 numbers from the user and print the smallest one.

9. Read 2 positive numbers from the user. Calculate the difference between the two of them (the absolute value).

   - If the diff variable is smaller than both values, print that those numbers are relatively-close (i.e. – num1=5, num2=9 then diff=4 => relatively-close!)
   - Validate that your values are numbers (hint: google something like: '*javascript check if number*')

10. Ask the user how many friends he has on FB and print out an analysis:

    - More than 500 – 'OMG, a celebrity!'
    - More than 300 (and up to 500) – 'You are well connected!'
    - More than 100– 'You know some people...'
    - Up to 100 – 'Quite picky aren't you?'
    - 0 – 'Let's be friends!'

11. Rolling Project: BankSystem

    - Initialize a variable: currBalance with the value: 1000
    - Prompt the user to get a secret pin code, '0796'.
    - After it was validated to be true, ask the user how much would he like to withdraw. Print a nice message with the new balance.
    - If the code was wrong, alert with a different message, and don't let him to withdraw the sum.
    - Add a feature: don't let the user withdraw more than he has in his account.

12. Guess Who

    - Use the alert function, and ask the user to think about an actor
    - Use the confirm function and ask the user 2 yes/no questions:
      Question 1: Is he a man?
        - Yes:
          - Question 2: Is he Blonde?
            - Yes: Philip Seymour!

        - No: Tom Cruise!
    - No:
        - Question 2: Is she English?
            - Yes: Keira Knightley!
            - No: Natalie Portman!

13. The Elevator

- Keep a currentFloor variable, initialize it to 0
- Ask the user which floor would he like to go to.
- Validate the floor is between -2 and 4.
- Update the currentFloor variable accordingly.
- Let the user know his current floor.
- If the user goes to floor 0 alert 'Bye Bye'.
- If the user goes to the parking lot (negative floors) alert: 'Drive Safely'.

Functions

14. Write a function that gets a user name as a parameter and greets the user.

15. Write a function that gets 2 numbers and returns their sum.

16. Write a function named *isEven* that gets a number, and returns true if the number is even. Otherwise the function will return false.

17. Write a function named *getBigger* that gets 2 numbers and returns the bigger one.

18. Write a function named *isAbove18* that gets a name and an age. This function checks if the age is above eighteen.
    In case the user is younger than eighteen, alert 'You are too young', otherwise alert 'You're allowed to buy a beer' (Use the user's name within the alerts).
    Also, the function should return a boolean value.
    * For now, we will make no use of the returned value.

Loops

*** Reminder: add ‘use strict’ at the top of your JS files.**

19. Read 10 numbers from the user, if the number is even, print it, otherwise print that the number is odd.

20. Read 10 numbers from the user and print:

    a. The maximum number.
    b. The minimum number.
    c. The average.

21. Read numbers from the user, until the number 999 is entered. For each number:

    a. Print if it's divided by 3.
    b. Print whether this number is much bigger (more than 10) than the previous number.

22. *+UnitTesting* Write a function named myPow that gets 2 parameters: base, exponent and returns the power. (use a loop…)

23. Write the function getFactorial that gets a number and returns n! (Google factorial if you are not sure what is the mathematical definition of it).

24. *+UnitTesting* Play with the function Math.abs(), read the documentation in MDN. Implement myAbs(), write the function yourself.

25. Write A function named getRandomInteger(min, max). The function should generate a random integer between the min and max parameters.
    Hint: Use Math.Random & Math.Floor.

    a. After you've played with it enough, read this page. Look at the *getRandomInt* function.
    b. Yes, it's better, now remember you can always use it later on in the course. (how amazing is that?)

26. Write a program that generates 10 random numbers. The numbers should be generated so each number is greater than the previously generated number.
    To simplify, generate the first number n so it is between (0→1000), and each subsequent number will be in the range of (n+1 → n+1001).
    example:
    First random number: (0→1000) 100
    Second random number: (101→1101) 748
    Third random number: (749→1749) 1650…

27. Asterisks!

a. Write the function *getAsterisks*(length) that returns a string containing asterisks according to the number supplied.
   For example: when the requested length is 4, it returns '****'
b. Write a function named: *getTriangle*(height) that returns a triangle:

(the parameter value here is 4)
Hint: use the function getAsterisks in a loop. Also, use '\n' to create a new line.

c. Write a function named: getMusicEqualizer(rowsCount) that generates random numbers between 1 and 10 and return columns in random lengths:

*****

**

***

***

d. Write a function that returns a block of asterisks (*) by the following parameters: rowsCount and colsCount. I.e: for 4, 5

*****

*****

*****

*****

Now, return only the outline:

*****

* *

* *

*****

e. Surprise, there is a new requirement to support any character (not necessarily asterisk), how easy would it be to refactor your code? The character should be decided by the user

28. Write a program that computes the greatest common divisor (GCD) of two positive integers.
    Example: 6, 15 => gcd: 3
    Hint: we need something like a loop: i 6 -> 2 and check modulus.

29. Read a number from the user (keep it as string such as “24367”) and then:

    a. Basic operations:

        i. Print each of its digits in a separate line.
        ii. Calculate the sum of its digits.
        iii. Calculate the multiplication (מכפלה) of its digits
        iv. Sum it’s first and last digits.
        v. Print it with it’s first and last digits swapped (2731=>1732)
        vi. Check whether it’s symmetric (like this number: 95459)
        vii. Print the number reversed (BONUS: as a number and not as string).

    b. BONUS: Special Numbers

        i. Check if the number is an *Armstrong number*. I.e 371 is an Armstrong number: *3*3+7*3*+*1*3 =371. If the number passed the test, print it to the console.
        ii. Check if the number is a *Perfect number*. Perfect number is a number that the sum of all its dividers is the number itself. I.e 6 is a perfect number (1+2+3).

iii. Read a number from the user. Store it in a variable called *max*. The function should print all the perfect numbers and all the *Armstrong numbers* that are smaller than *max*.

Strings

30. Read 2 names from the user and print the longest.

31. Read a string from the user and print:

    c. Its length.
    d. Its first and last characters.
    e. The string in uppercase and lowercase letters.

Strings and Loops

32. Read a string from the user and print it backwards using a loop.

33. *+UnitTesting* VOWELS (aeiou)

    code the following functions:

    a. Write a function named *printVowelsCount(str)* that gets a string and print how many times each vowel appears.
    b. Write a function that gets a string and changes the vowels to lowercase letters, and the rest to uppercase letters (GiZiM GiDoo).
    c. Write a function that gets a string and doubles all the vowels in it.

    Test the functions using the inputs: “aeiouAEIOU” “TelAvivBeach”

34. *+UnitTesting* write a function named *myIndexOf*(str, searchStr) that accepts 2 strings. The function returns the index of the second string in the first, if it wasn’t found, return -1 (don’t use the built-in indexOf...).

35. *+UnitTesting* Write the function encrypt that gets a string and encrypts it. It replaces each character code with the code+5 (I.e. 'r' will be replaced by: 'w'). NOTE: The function should encrypt the entire string by shifting each letter as described above. Now write the function decrypt that decrypts a message. Tip: try to write in the console: 'ABC'.charCodeAt(0)
    Tip - search for the opposite function to charCodeAt

Bonus: extract the common logic to an *encode* function that both encrypts and decrypts.

36. *+UnitTesting* Write a function that gets a string of names delimited by a comma. I.e: 'igal,moshe,haim' and prints the longest name, and the shortest name. Tip: use the function indexOf, note that the function accepts 2 parameters

37. Write a function named *generatePass*(passLength) that generates a password of a specified length. The password is made out of random digits and letters.

Arrays

38. Write a function named *biggerThan100*. It gets an array of numbers and returns an array of only the numbers that are bigger than 100.

39. Write a function named *countVotes*(votes, candidateName) that counts how many votes this candidate got. i.e.: if the votes array looks like this: ['Nuli', 'Pingi', 'Uza', 'Shabi', ‘Uza’] And the candidateName is : 'Uza', the function returns 2.

40. Write a function named *getLoremIpsum*(wordsCount) that return a sentence with random dummy text (google: lorem ipsum...) TIP, here are the steps you may use to solve this:

    a. First, write a function named *getWord()*. The function returns a single word made out of 3-5 random letters the length of the word will be generated randomly. Tip: you can create a string or an array of all the characters in the English alphabet.

    b. Lastly, call this function in a loop to create a sentence.

41. *+UnitTesting* Write a function named *sayNum*(num) that prints each digit in words. I.e: 123 => One Two Three. 7294 => Seven Two Nine Four. TIP: You may use Switch inside a loop OR an array named digitNames. (Or what the heck, try them both.)

42. Write a function named *startsWithS* that gets an array of names and returns an array of the names that start with S.

    Step2: Refactor your function to work on any letter by adding a letter parameter, you might need to rename the function so it will suit it’s new functionality.

43. Write the function *sumArrays* that gets 2 arrays and returns the sum of the two arrays. I.e: [1, 4, 3] [2, 5, 1, 9] => [3, 9, 4, 9]

    TIP: this can be done in a single loop by first identifying the shorter or longer array from the two.

    Step2: Read these arrays from the user (until the number 999 is entered) TIP: write the function: *getArrayFromUser* and call it twice

44. Write the function *printNumsCount*(nums). The array nums will contain integers in the range of 0-3 such as:

3	2	0	2	2	0	3

    The function prints how many times each of these numbers appears in the array.

    GUIDANCE: the fact that the values are in a specific range allows us to use an array where the index is actually the number itself. The value in the array counts the appearances of the numbers.

    e.g. for the array [3,2,0,2,2,0,3] the array will look like this: [2,0,3,2]

45. Write the function *removeDuplicates*(nums). The array nums should contain numbers in the range of 0-10 such as:

5	4	5	7	1	9	4

    the function returns a new array in which each value appears only once (e.g. in this case: 5, 4, 7, 1, 9)

    TIP: Notice that the values are in a specific range.

46. *+UnitTesting* Write the function: *multBy*(nums, multiplier) that returns a modified array in which each item in the array is multiplied by a multiplier.

    Step2: Add another param: isImmutable. It will be a variable that when it's value is set to true, use array.slice() to work on a new array. Leave the original array as it was.

47. Implement your own version of the split function: *mySplit*(str, sep) Test it with different types of strings and separators. I.e 'Japan,Russia,Sweden'

    You can assume that the separator (delimiter) is a single character. (BONUS: don't assume that)

48. *+UnitTesting* Write the function *getNthLargest*(nums, nthNum) to get the nth largest element from an array of unique numbers. I.e: *getNthLargest* ([7, 56, 23, 88, 92, 99, 89, 11], 3) Result: 89

    a. It will be easier if the array is sorted first.
    b. BONUS: Try writing the algorithm without sorting the array.

49. Implement the function *sortNums*(nums) this function returns a sorted array (without changing the given array).

    It works by looping through the array, finding the minimum, splicing it out, and adding it to the new array.

    Read about how to sort an array yourself, by using the **bubble sort** algorithm, google it, copy it and use it.

50. Making Water! Let's imagine that we have the following atoms:

1	Hydrogen	H
5	Boron	B
6	Carbon	C
7	Nitrogen	N
8	Oxygen	O
9	Fluorine	F

    a. Use an array with letters that stands for each atom.
    b. Pick random atoms from the array to create molecules of 3 atoms.
    c. Stop when you got 'HOH' for water. (Two Hydrogens and one Oxygen)
    d. Print how many tries you had before 'HOH' was drawn.

Objects

51. *+UnitTesting* **Object as a Map**: Write the function *countWordApperances*(txt) that returns an object map. This object will have a key that will be the word. The value will be the count (how many times this word appeared in the string).
    example: *countWordApperances*('puki ben david and muki ben david') will return:
    { puki: 1, ben: 2, david: 2, and: 1, muki: 1 }

52. **Monsters**:

Create an array of monsters with 4 monsters (use a *createMonsters*() function)

a. Each monster should also have
   i. id – a unique sequential number
   ii. name – that you will read from the user
   iii. power (random 1-100)

b. Write the functions:
   i. *createMonsters*()

   ii. *createMonster*(name, power) – returns a new monster object. The name and power parameters are optional. That means that you should set them to a defaultive value if nothing is sent in the parameters.
   iii. *getMonsterById*(id) - finds and returns a monster object by its id.
   iv. *deleteMonster*(id) - the function removes the specified monster from the array.
   v. *updateMonster*(id, newPower) - the function updates the specified monster, setting a new power.

c. Write the function: *findMostPowerful*(monsters).

d. Write the function: *breedMonsters*(monsterId1, monsterId2), the function returns a new monster. The breeded monster power is an average of its parents power. The name is the beginning half of the first parent name, and the second half is the end of the second parent name.

53. **Students:**

a. Create a students array (use the same algorithm as before and name it createStudents())
b. Read the student name from the user until “quit” is entered. Populate the students array with student objects.
c. Read 3 grades for each student (each student should have a grades array).
d. Calculate the average of each student.
e. Write the function *findWorstStudent(students).*
f. Write the function *sortStudentsByGrade(students)*.
g. Write the function *sortStudentsByName(students)*

54. **Airline:**

a. Build a data structure for an airline company. (use the create function for each object). Create to following entities:

   i. A Plane. The plane will contain:
      1. model.
      2. seatCount.

   ii. A passenger - tip: use *createPassenger*(fullName, flights)
      1. id (7 random digits)
      2. fullName
      3. flights (array of pointers to the relevant flights)

   iii. A flight
      1. date
      2. departure
      3. destination
      4. plane (pointer to a plane)
      5. passengers (array of pointers to the relevant passengers)

b. Initialize all variables with consistent data. I.e (date should be 0 and passengers should be an empty array).

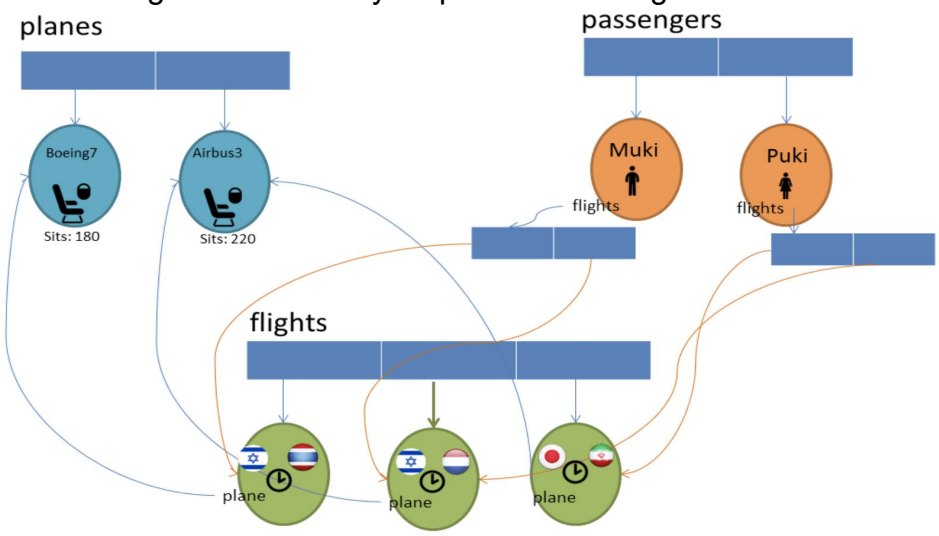
   i. Create an array of 5 passengers (gPassengers is a good name)
   ii. Create an array of 2 planes.
   iii. Create an array of 2 flights, each flight has a plane property that points to a plane object, and a passengers property that points to the passengers array.

   TIP: first create a passenger with an empty *flights* array, and the flight with an empty *passengers* array, then you can push the objects.

c. Write the functions:

   i. *bookFlight*(flight, passenger) - this function connects between the pointers of the passengers and their flights.
   ii. *getFrequentFlyers*() - returns the passengers with the maximal flights count.
   iii. *checkIfFlightFullyBooked*(flight) - checks if there are available seats on the flights, and returns true if there are. Think where would it make sense to invoke it.

The following illustration may help understanding the data structure:



Multi-Dimensional Arrays

55. Fill up a multi-dimensional array with numbers, and then, Write the following functions:

    a. sumCol(mat, colIdx)
    b. sumRow(mat, rowIdx)
    c. findMax(mat, colIdx)
    d. findAvg(mat)
    e. sumArea(mat, rowIdxStart, rowIdxEnd, colIdxStart, colIdxEnd)

56. Symmetric Matrix:

    A symmetric matrix is a matrix that passes this boolean condition:

    mat[i][j] === mat[j][i]

    Write the function *checkIfSymmetric*(mat).

57. Write the function *findMode*(mat) that will print out the number that appears most frequently in the multi-dimensional array.

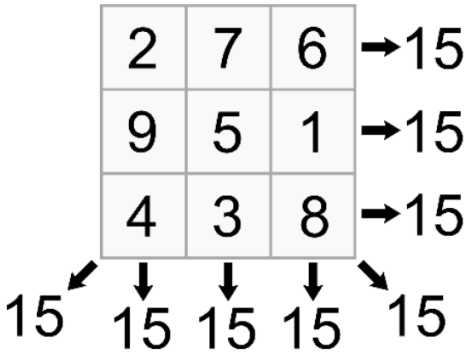
    BONUS: If there are ties (e.g.: both 47 and 53 appeared 17 times), print both of them, or all of them. (TIP: use an object map to count the numbers)

58. Write a function that gets a 2d array and vailates it's a magic square:

    a. It must be a square

b. The rows, columns, and the two diagonals sums should be equal.

Example:



59. **Bingo** - Your challenge is to simulate a Bingo game. In this game there are 2 players. Each player has his own board (with different random numbers). The winner is the player who marks all the numbers on his board first. Here is the suggested data structure:

```
var gPlayers = [
    { name: 'Player1', hitsCount: 0, board: createBingoBoard(5) },
    { name: 'Player2', hitsCount: 0, board: createBingoBoard(5) }
];
```

```
// every cell in the mat will hold an object like this:
{ value: 78, isHit: false }
```

Development Plan

a. Implement the *createBingoBoard(size)* function
   i. Start by implementing a function that returns a 5*5 matrix containing cell object as described above with the numbers 1..25

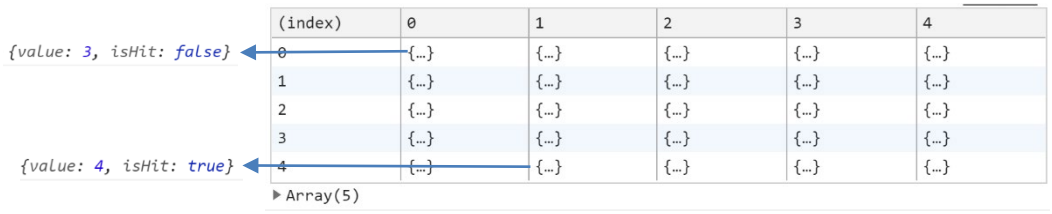
   ii. Implement the function *printBingoBoard(board)* that prints the board showing only the number (value) in each cell.
      1. If the isHit value in the object is true, add 'v' to the printed number.

      2. Check your function by manually setting a cell's isHit to true:
         gPlayers[0].board[0][0].isHit = true

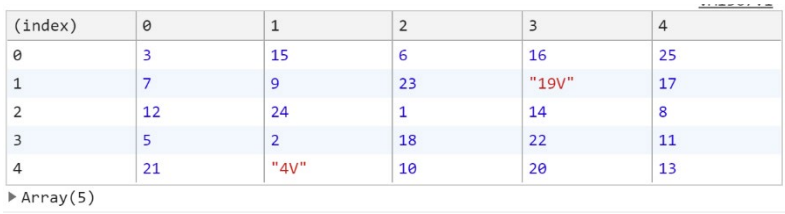
*after you see that it works you can comment out or delete this line

and print the board.

Now you should have two boards.
The first board contains the data objects. This board was created by the **createBingoBoard** function. If you print this board to the console you will see something like this:



The second board is just for the player to see the number in each cell (or the number + 'v' if the cell isHit). It is not a part of our data model but our way to view the game. This board was created by the **printBingoBoard** function**.**
When you print this board to the console you will see something like this:



b. Implement the *playBingo* function:

```
function playBingo() {
    var isVictory = false;
    while (!isVictory) {
        var calledNum = drawNum();
        for (var i=0; !isVictory  && i < gPlayers.length; i++) {
            var player = gPlayers[i];
            markBoard(player, calledNum);
            isVictory = checkBingo(player);
        }
    }
}
```

i. It is a common practice to setup the structure and add the implementation in steps, in this case:
   1. *drawNum* can be a simple function for now returning a fixed number (e.g. 17)
   2. *markBoard* can be an empty function for now.

3. checkBingo can be a simple function returning true for now (note, if the function will return false it will be in an infinite loop).

c. Now we can implement the *markBoard* function:
    1. If there is a cell with the calledNum, update the cell's isHit value accordingly and also increase the player hitsCount .
    2. Use the *printBoard* function to debug your function and make sure it works correctly.
d. Implement the *checkBingo* function:
    1. Just check if the player *hitsCount* has reached 25.
e. Implement the *drawNum* function:
    1. We will later need this function to return a random number, but we don't want repetitions, so we will use an array - gNums.
    2. Add the function *resetNums* that updates the global variable: gNums to be an array with the numbers 1..25
    3. This function should be called at the beginning of *createBoard* and also at the beginning of the *playBingo* function.
    4. The function drawNum can just *pop* from that array for now (predictable order helps while developing)

f. At this stage you should have a basic working game that ends after 25 iterations. Do you?
g. OK so now implement the following additions/modification:
    i. The gNums array should have numbers from 1 to 99.
    ii. *drawNum* should return (use splice) a random number from the array.
    iii. Print a happy greeting when a player:
        1. completes a row: 'Muki has completed a row!'.
        2. completes the main diagonal: 'Muki has completed the main diagonal!'
        3. Completes the secondary diagonal 'Muki has completed the secondary diagonal!'.
    iv. Slow down the game so it feels more realistic and easy to follow:
        1. Use setInterval instead of the while loop:
            a. var gameInterval = setInterval(playTurn, 1000)
        2. User clearInterval(gameInterval) when the game is over.

60. **Game of Life**

המשחק אמור לתאר לידה ומוות של יצורים, כאשר הלידה והמוות נקבעים לפי כללים מסוימים.
שדה המשחק: לוח משבצות שגודלו נקבע על פי השחקן (המשתמש).
יצור חי יתואר ע״י X. אם אין יצור חי המשבצת תישאר ריקה.
כל התאים הסמוכים למשבצת מסוימת הם השכנים שלה (כלומר לכל היותר 8 שכנים).

			X	X			X
X		X		X	X	X	
			X	X		X	
	X						
					X		
		X		X	X		
X	X		X		X	X	

כללי המשחק:

1. במשבצת שלה 0-2 שכנים שהם יצורים חיים, לא יתכנו חיים בדור הבא (כלומר: אם היו בה חיים היצור ימות מבדידות).
2. במשבצת שלה 3-5 שכנים שהם יצורים חיים, יתכנו חיים בדור הבא (כלומר: אם לא היה בה יצור – ייוולד חדש, ואם היה – הוא ישאר בחיים).
3. במשבצת שלה 6-8 שכנים שהם יצורים חיים, לא יתכנו חיים בדור הבא (כלומר: אם היו בה חיים היצור ימות מצפיפות).

טיפ, הרץ פונקציה בסגנון זה באמצעות אינטרבל:

```
function play() {
    gBoard = runGeneration(gBoard)
    renderBoard(gBoard)
}
```

הפונקציה מקבלת לוח ומחזירה לוח חדש בו המצב המעודכן

טיפ: יש לבצע את השינויים במטריצה חדשה כדי לא לקלקל את הקיימת תוך כדי החישוב