Javascript Basics

The 60 Exercises

Setup

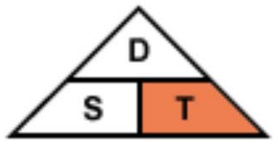
- Use the same exercise-runner project for all exercises below.
- For each exercise, create a different JavaScript file (01.js, 02.js…).
- Copy the assignment as a comment to the top of your JavaScript file.
- At the top of your JavaScript file use ***`console.log()`*** to print the number and purpose of the exercise to the console – i.e. “Ex 03: Convert Celsius to Fahrenheit”.
- The output printed to the console should be easy to understand and should follow the following pattern wherever possible:

```
INPUT:      2, 5
EXPECTED:   The biggest number is: 5
ACTUAL:     <Activate your code here to see how it compares with your expectations>
```

In some exercises (marked with TEST) we will specifically request unit testing, try to think about different INPUTs to make sure code behaves as expected.

Basics

1. Use ***`prompt()`*** to read 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 *use* ***`console.log()`*** to print 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.



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

5. Read 3 digits from the user, join them to one number and print it:
   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.

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

   a. Print the values of the following calculations to the console:

      - `-b`
      - `2 * a`
      - `b*b - 4*a*c` (this is called the **discriminant)**

   b. BONUS:

      a quadratic equation looks like this:

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

      The two solutions for of this equation are X_1 and X_2:

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

      - Print the quadratic equation as a string to the console
      - Print the solutions of X_1 and X_2 to the console.
      - If the **discriminant** is 0 – the equation has a single solution
      - If the **discriminant** is negative – the equation has no solutions

      Here's an 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:	***2X² – 5x + 2 = 0*** ***x1 = 2 ; x2 = 0.5***

Hint: To print the ***x²*** 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 it is, print all the numbers to the console like this: ***6 + 4 = 10***
   If not, print them like this: ***3 + 5 != 10***

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

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

   - First, check that the input values are numbers (try googling something like '*javascript check if number*')

   - Print the inputs and the absolute difference between them to the console.

   - If the difference is smaller than both input values, print to the console that those numbers are relatively close.
     Something like this –

     num1 = 5, num2 = 9, diff = 4 => numbers are relatively close!

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

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

11. Bank System

    - Initialize a variable named currBalance with the value: ***1000***
    - Initialize a constant named ***PIN*** with the value: ***'0796'***

- Prompt the user to enter a secret pin code.
- If the user entered the correct pin code, ask him how much he would like to withdraw. Print a nice message with the new balance.
- If the pin code was wrong, alert the user with a different message, and don't allow him make a withdrawal.
- 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!

- Print the answer to the console:

13. The Elevator

- Initialize a ***`currFloor`*** variable to 0.
- Ask the user which floor would he like to go to.
- Check that the floor is between -2 and 4.
- Update the ***`currFloor`*** 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'.

14. Write a function which receives a user name as a parameter and greets the user.
15. Write a function which receives two numbers and returns their sum.
16. Write a function named ***`isEven`*** which receives a number, and returns ***`true`*** if the number is even, and ***`false`*** if it is odd.
17. Write a function named ***`getBigger`*** which receives two numbers and returns the bigger one.
18. Write a function named ***`isAbove18`*** which receives a name and an age.
    The function should check if the age is above eighteen and use ***`alert`*** to show a message :

    If the user is younger than eighteen, the message will be 'You are too young',
    otherwise, the message will be David, You're allowed
    (Use the user's name within the alerts).

    The function should also return a boolean value, print it to the console.

Loops

Reminder: Always add **`'use strict'`** at the top of your JS files.

19. Read 10 numbers from the user. Check each number – if it is even, print it, otherwise print a message saying 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 is divisible by 3.
    b. If it is bigger by more than 10 from the previous number, print a suitable message.

22. Write a function named ***`myPow`*** which receives 2 parameters: base, exponent
    Sample unit testing:

    INPUT: 2, 3
    EXPECTED: 8
    ACTUAL: 8

23. Write the function ***`getFactorial`*** which receives a number and returns ***`n!`*** .
    (Google 'factorial' if you are not sure what the mathematical definition of it is).

24. Try playing around with the function ***`Math.abs()`*** and read its documentation on MDN.
    Implement a function called ***`myAbs()`***, which mimics the behavior of ***`Math.abs()`***.

25. Write A function named ***`getRandomInt(min, max)`***. The function should generate a random integer between ***`min`*** and up to, but not including, ***`max`***.
    *Tip: use **`Math.Random`** and **`Math.Floor`**.*

    After you've worked it out, read this page. Look at the implementation of the ***`getRandomInt`*** function.

26. Write a program which generates 10 ascending random numbers (each number is greater than the previously generated number).
The first number ***n***, should be in the range ***0 - 1000***,
and each subsequent number, should be in the range ***n+1 - n+1000***

For example:

First random number:	***0 - 1000***	**=>**	***100***
Second random number:	***101 - 1101***	**=>**	***748***
Third random number:	***749 - 1749***	**=>**	***1650***...

27. Asterisks!

a. Implement the function ***getAsterisks(length)*** which returns a string of asterisks. The number of asterisks in the string is determined by ***length***.
For example: when the requested length is ***4***, it returns ***'****'***

b. Implement the function ***getTriangle(height)*** which returns a triangle.
For example, ***getTriangle(4)***, will return a string which will look like this when printed to the console:

```
*
**
***
****
***
**
*
```

Hint: use the function ***getAsterisks*** *in a loop. Also, use* ***\n*** *to create new lines.*

c. Implement the function ***getMusicEqualizer(rowCount)*** which generates random numbers between 1 and 10 and returns rows of random lengths.
For example, ***getMusicEqualizer(5)*** , will return a string which will look something like this when printed to the console:

```
**
******
*****
***
*****
```

d. Implement the function ***getBlock(rows, cols)*** which returns a block of asterisks in the dimensions given by its parameters.
For example, ***getBlock(4, 5)*** , will return a string which will look like this when printed to the console:

```
*****
*****
*****
*****
```

Now Implement ***getBlockOutline(rows, cols)***, which only returns the block outline:

```
*****
*   *
*   *
*****
```

e. Surprise! There is a new requirement to support any character (not necessarily asterisks). The character should be decided by the user.
Refactor your code to support this requirement
How would you rename the function to better describe its new functionality?

28. Write a program which calculates the greatest common divisor (GCD) of two positive integers.
For example: if the input are ***6*** and ***15***, the calculation's result should be ***3***.
Tip: we need a loop that runs from 6 to 2 and checks the 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 its first and last digits.
   v. Print it with its first and last digits swapped. (***2731 => 1732***)
   vi. Check whether it is symmetric. (like this number: ***95459***)
   vii. Print the number reversed.
   viii. BONUS: Print the number reversed as a **number** and not as string.

b. BONUS: Special Numbers

   i. Check if the number is an *Armstrong number*. An *Armstrong number* is an integer, such that the sum of each of its digits, raised to a power equal to the number of its digits, is equal to the number itself.
   For example: ***371*** is an Armstrong number, because $3^3+7^3+1^3 = 371$.
   Another example: ***548834*** is an Armstrong number,
   because $5^6+4^6+8^6+8^6+3^6+3^6 = 548834$
   If the number passed the test, print it to the console.

   ii. Check if the number is a *Perfect number*. A perfect number is a positive integer that is equal to the sum of its divisors.
   For example: ***6*** is a perfect number ***(1+2+3)***.

   iii. Read a number from the user and store it in a variable named ***max***.
   Implement a function which will print all the *Perfect* numbers and all the *Armstrong numbers* that are smaller than ***max***.

coding academy

Strings and Loops

30. Read two names from the user and print the longer one.

31. Read a string from the user and print:

    a. Its length.
    b. Its first and last characters.
    c. The string in all uppercase and then, in all lowercase letters.

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

33. VOWELS (aeiou)

    Implement the following functions:

    a. Implement a function named ***`printVowelsCount(str)`*** which receives a string and prints the number of occurrences of each vowel in it.
    b. Write a function which receives a string and changes all its vowels to lowercase letters, and all other letters to uppercase. For example: (***`Upset  =>  uPSeT`***).
    c. Write a function which receives a string and doubles all the vowels in it.
       Test the functions using the inputs: ***"aeiouAEIOU"*** and ***"TelAvivBeach"***

34. Implement a function named **`myIndexOf(str, searchStr)`** which receives two strings. The function returns the index of the second string within the first, or ***`-1`*** if it wasn't found (do not use the built-in ***`indexOf`*** function…).

35. Implement the function ***`encrypt(str)`*** which receives a string and encrypts it. It does so by replacing each character ***`code`*** with ***`code + 5`***.
    For example: ***`'r'`*** will be replaced by: ***`'w'`***.

    Now implement the function ***`decrypt(str)`*** which decrypts a string previously encrypted by ***`encrypt()`***.

    Tip: try running this in the console: ***`'ABC'.charCodeAt(0)`***
    Tip: search for the opposite function of ***`charCodeAt`***

BONUS: extract the common logic to an ***`encode`*** function, which does both encryption and decryption.

36. Implement a function which receives a string of comma separated names, for example: ***`'Igal,Moshe,Haim'`***, and prints the longest name and the shortest name.
Tip: use the function **`indexOf`**. *note that the function accepts two parameters.*

37. Implement a function named ***`generatePass(passLength)`*** which generates a password of the specified length. The password is made out of random digits and letters.

Arrays

38. Implement a function named ***biggerThan100***. It receives an array of numbers and returns a **new** array containing only the numbers which are greater than ***100***.

39. Implement a function named ***countVotes(votes, candidateName)*** which counts how many votes a candidate received.
For example: if the ***votes*** array looks like this:

    ['Nuli', 'Pingi', 'Uza', 'Shabi', 'Uza']

    And the ***candidateName*** is : ***'Uza'***,

    the function returns ***2***.

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

   a. Create a string or an array of all the characters in the English alphabet.
   b. Write a function named ***getWord()*** which returns a single word made of 3 - 5 random letters. The length of the word will be generated randomly.
   c. Call this function in a loop to create a sentence.

41. Write a function named ***sayNum(num)*** which prints each digit in words. For example:

```
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***, which receives an array of names and returns a new array, containing only those names which begin with ***S***.

    Now, refactor your function to work on any letter by adding a letter parameter. Rename the function to reflect its new functionality.

43. Write the function **`sumArrays`** which receives two arrays and returns their sum.

    For example: ***`[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 of the two.

    Now, 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 parameter ***`nums`***, is an array of integers in the range ***`0 - 3`*** like this one:

3	2	0	2	2	0	3

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

    Tip: the fact that the values are in a specific range allows us to use a second array, in which the index, is actually the number itself.
    *The values of this second array, will store the occurrences of the numbers in **`nums`**.*

    For example: INPUT: ***`[3,2,0,2,2,0,3]`***
    EXPECTED: ***`[2,0,3,2]`***

45. Write the function ***`removeDuplicates(nums)`***.

    The parameter ***`nums`***, is an array of integers in the range ***`0 - 5`*** like this one:

5	4	5	2	1	2	4

    For example: for this input array ***`[5,4,5,2,1,2,4]`***
    the output will look like this: ***`[5,4,2,1]`***

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

46. Write the function: ***`multBy(nums, multiplier)`*** which modifies the ***`nums`*** array so that each of its items, is multiplied by ***`multiplier`***.
    The function returns the modified array.

    Step2: Add another parameter to this function named ***`isImmutable`***.
    When this parameter is ***`true`***, use ***`array.slice()`*** to perform the function's calculation on a copy of the array, and leave the original array unchanged.
    The function should return the modified array.

coding academy

47. Implement your own version of the ***`split`*** function - ***`mySplit(str, sep)`*** .
    Test it with different types of strings and separators.
    You can try – ***`'Japan,Russia,Sweden'`*** or ***`'1-800-652-0198'`***

    You can assume that the separator (delimiter) is a single character.
    BONUS: don't assume that, e.g: ***`'A|||B|||C'`***

48. Implement the function ***`sortNums(nums)`*** . This function returns a sorted version of the original ***`nums`*** array. It works by looping through the array, finding the minimum, splicing it out, and adding it to a new array.

    Read about how to sort an array yourself, by using the bubble sort algorithm. Google it, copy it, adjust it to match our coding conventions and use it.

49. Write the function ***`getNthLargest(nums, nthNum)`*** to get the nth largest element from an array of unique numbers.
    For example: ***`getNthLargest([ 7, 56, 88, 92, 99, 89, 11], 3)`***
    will return: ***`89`***

    Tip: This can be done more easily by first sorting the array.
    BONUS: Try writing the algorithm without sorting the array.

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. Create an array of letters, each one representing one type of atom from the above list.
b. Pick random atoms from the array to create molecules of 3 atoms.
c. Stop when you get 'HOH' (Water – two Hydrogen atoms and one Oxygen atom).
d. Print the number of attempts it took to get 'HOH'.

Objects

51. Object as a Map
    Implement the function ***countWordApperances(txt)*** which returns an object map.
    The object keys will be the words in the string.
    The values will be the number of times each of these words appears in the string.
    For 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 object should have the following keys –

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

    b. Implement the following functions:

        i. ***createMonsters()***
        ii. ***createMonster(name, power)*** – returns a new monster object. The name and power parameters are optional – set them to default values if they aren't passed to the function.
        iii. ***getMonsterById(id)*** – scans the array for a monster with the provided ***id*** and returns it.
        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 a 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 grade for 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.

      For each type of object, implement a *create* function.

      Create to following entities:

      i. A Plane –
         Tip: implement **`createPlane(model, seatCount)`**
         The plane will contain the following keys:
         1. **`model`**
         2. **`seatCount`**

      ii. A passenger –
         1. **`id`** – composed of 7 random digits
         2. **`fullName`**
         3. **`flights`** – an array of pointers to the relevant flight objects

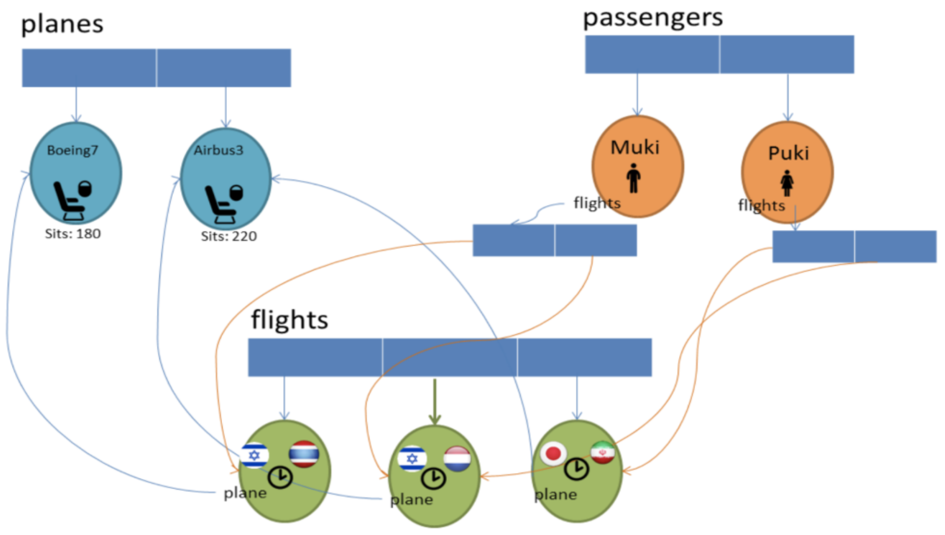
iii. A flight
   1. ***date***
   2. ***departure*** – where the flights takes off from…
   3. ***destination*** – …and where it lands.
   4. ***plane*** – a pointer to a plane object
   5. ***passengers*** – an array of pointers to the relevant passenger objects

b. Initialize all variables with consistent data. ***date*** should be 0, ***passengers*** should be an empty array, etc…
   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 should have a plane property that points to a plane object, and a passengers property that points to a passengers array.

   TIP: first create a passenger with an empty flights array, and the flight with an empty passengers array, and then connect them.

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. ***isFullyBooked(flight)*** - checks if all seats are booked, and returns true if they are. Where would it make sense to invoke this function?

The following illustration may help understanding the data structure:


planes
passengers
Boeing7
Sits: 180
Airbus3
Sits: 220
Muki
flights
Puki
flights
flights
plane
plane
plane

2D Arrays - Matrix

55. Fill up a matrix 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. In statistics, the most common value in a set of data, is called the mode (שכיח).
    Write the function ***`findMode(mat)`*** . The function prints the number which appears most frequently in a matrix.

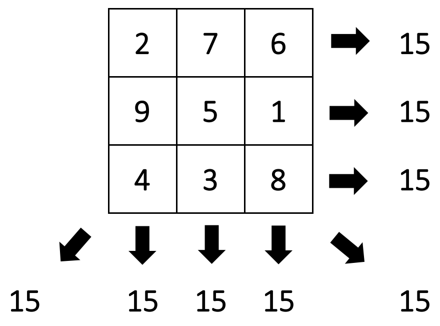
    BONUS: If there are ties, e.g. both 47 and 53 appeared 17 times, print both of them, or all of them.

    Tip: use an object map to count the numbers.

58. Write a function which receives a 2D array, and tests whether it is a magic square:

   a. It must be a square

   b. The sums of the rows, columns, and the two diagonals should all be equal.
      For example:



59. BINGO - Your challenge is to implement the famous game of Bingo. In this version of the game, there are few players. Numbers are drawn at random, and each player marks the number if it appears in his board. The first player to mark all the numbers on his board, wins.

    Here is the suggested data structure:

```
var gPlayers = [
     {name: ‘Muki’, hitCount: 0, board: createBingoBoard()},
     {name: ‘Puki, hitCount: 0, board: createBingoBoard()}
]
```

    Every cell in ***board*** will hold an object like this:

```
{value: 17, isHit: false}
```

    It is common practice to implement our code in several stages, starting with simplified version for some of the functions, which allow us to test other parts of our code first. Let’s try this approach:

   a. Implement the ***createBingoBoard()*** function:
      It returns a 5*5 matrix containing cell object as described above,
      with the numbers 1 – 25 (Later on we will change it to 1-99).

b. Implement the function ***printBingoBoard(board)*** which prints the board showing just the number (value) in each cell.

   If ***isHit*** is true, add 'v' to the printed number.

   Check your function by manually setting a cell's *isHit* to true such as: *gPlayers[0].board[0][0].isHit = true* and printing the board.

   (TIP: remember you can run code from the console)

   i. Implement some empty functions:

      1. ***drawNum()*** code a simple function returning a fixed number (e.g. 17)

      2. ***markBoard()*** an empty function for now.

      3. ***checkBingo()*** simple function returning ***true*** (note, that if it returns ***false*** it will cause an infinite loop).

c. 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);
        }
    }
}
```

d. Implement the ***markBoard*** function:

   i. If the board contains a cell with ***calledNum*** in its ***value***, update that cell's ***isHit*** value accordingly and increase the player's ***hitCount***.

   ii. Use the ***printBingoBoard*** function to debug your function and make sure it works correctly.

e. Implement the ***checkBingo*** function – Just check if the player's ***hitCount*** has reached 25.

f. Implement the ***drawNum*** function:

   i. We will later need this function to return a unique random number, so we will use an array - ***gNums***.

ii. Add the function ***resetNums*** which updates the global variable ***gNums*** to be an array with the numbers 1 – 25. This function should be called at the beginning of ***createBoard*** and also at the beginning of the ***playBingo*** function.

iii. The function ***drawNum*** can just ***pop*** from that array for now (predictable order helps while developing)

g. Make sure you have a basic working game that ends after 25 iterations before moving on

h. Implement the following additions and modification:

i. The ***gNums*** array should hold the numbers from 1 to 99.

ii. ***drawNum*** should return a random number from the array. Use ***splice*** for that, to make sure the drawn numbers do not repeat.

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:
var gameInterval = setInterval(playTurn, 1000)

2. Use ***clearInterval(gameInterval)*** when the game is over.

i. Finalizing

i. How easy it is for your game to work for a 6*6 bingo board?

ii. How easy it is to add more players?

60. The **Game of Life** is a simulation of how a population of creatures evolves from one generation to the next, based on a set of simple rules.
This colony is described by a matrix of a user determined size, where each cell is either populated by a creature (marked by an asterisk **`'*'`**), or vacant.
As with any matrix, each cell can have 8 neighboring cells at the most.

Start with a population of your choice, for example:

*						*			
					*	*			
			*						
					*				
				*		*			

Here are the rules which govern the evolution of the colony:

a. if a creature has 0 – 2 neighboring creatures, it will die of loneliness and the cell which it occupies will become vacant in the next generation.

b. if a cell has 3 – 5 occupied neighboring cells, it will have a creature in it in the next generation – either a newborn creature or the creature which previously occupied it.

c. if a creature has 6 – 8 occupied neighboring creatures, it will die of suffocation and the cell which it occupies will become vacant in the next generation.

Tip: use ***`setInterval`*** to run a function which looks something like this:

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

Tip: use a second matrix to generate the new generations of the colony so that you do not modify the current state of the colony while still calculating the next generation.