סיכומים.

ב-git

אם רושמים

gic clone https://....git זה מוריד את הפרויקט למחשב

ואז אם רושמים ב-DOS code זה פותח את זה מיד בתוך ה-VSC.

CSS/HTML

שבוע 1

-דוגמאות לאלמנטים לא סמנטיים: <div> ו-<span> - לא אומר דבר על התוכן שלו.   
דוגמאות לאלמנטים סמנטיים: <form>, <table>, ו-<article> - מגדיר בבירור את התוכן שלו.

האלמנט <section> מגדיר קטע במסמך.

בשימוש בעיקר ל -

פרקים

מבוא

ידיעות חדשות

פרטי התקשרות

האלמנט <header> מייצג מיכל לתוכן מבוא או קבוצה של קישורי ניווט.

רכיב כותרת אחד או יותר (<h1> - <h6>)

לוגו או אייקון

מידע מחבר

האלמנט <footer> מגדיר כותרת תחתונה למסמך או למקטע.

**שבוע 2**

התשובה המדוייקת לשאלה שנשאלתי היום בבוקר בהרצאה, ולא היתה לי תשובה בשלוף, היא שההבדל בין align לבין justify ב-flex הוא ש-justify מיישר אלמנטים על ה-main-axis ו-align מיישר אלמנטים על ה-cross-axis

כל חלק שרוצים לשים ב-flex- צריך לרשום בתוכו שהוא display:flex

אם מגדירים space-between בקונטיינר וגם ממורכז ב- justify-contenct

אז מה שיקרה בבנים, שיהיה רווח בין שני, וכל אלמנט ילך הצידה, אם נניח שמנו

שניים אז אחד יהיה בימין ואחד בשמאל ובאמצע יהיה רוחח.

לכל רכיב אפשר להגדיר:

flex grow- הגדלה

flew shrink- הקטנה

flex-basis - גודל בסיסי לתיבה

flex 0 0 200px

קיצור של cntl+/ מאפשר לסמן את הקוד ולעשות שיהיה comment

Padding – זה ברמת הצורה.

Margin – מחוץ לצורה ביחס למסך.

אחרי float עושים clear.

justify-content - עובד על ציר X , בדכ שמים באמצע.

שינוי ברירת המחדל של rem

עושים בבודי font-size:6.25%; יגרום לזה להיות ביחס 1:1.

ברירת מחדל זה יחס של 1:16.

מיקומים - סטטי - לא ניתן לשנות מיקומים במסך,

רלטבי - מאפשר לבנים להיות אבסולוט ביחס אליו, .[

תמיד בן של האב, זה אבסולוטי ביחס לאבא בלבד.

**איך עושים מירכוז תיבה:**

-מיקום אבסוליטי הולכים 50 אחוז שמאלה וגם למעלה.

ואז עושים transfrom: translate(-50%,-50%);. ב-transfrom - יש בגלל הזזה, הגדלה, סיבוב וכו'.

-אפשרויות נוספות למיקום מרכזי:

display:flex; justify-content:center; aligen-items:center

ב-Layout-ים מורכבים יותר יהיה קל יותר לעשות את זה עם flaex ושיטת אבסולוט זה עדיף רק למשהו סציפי ולא ריבוי אלמנטים שם זה מסובך יותר.

-שיטה נוספת היא grid שנלמד בהמשך שמאפשרת לעשות את זה גם כן.

-יש גם ע"י float אבל לרוב לא יהיה לזה שימוש כי זה רק בלאגן עושה.

-מגדירים ברירת מחדל Border-box בתוך בודי שלא יהיו שינויים

משהו שישפיעו.

before ו after - בד"כ זה יהיה רלטיבי באלמנט, ושמה ב- before ואפטר נשים אבסולוטי

ונשים 0 במיקומים שלהם ונשלחק את זה (מיקום כמו left וכו').

Before - אם רוצים לפני או מאחורי האלמנט (כלומר בדף האחורי).

after -זה יהיה מעל האלמנט כמו דף שנמצא על דף אחר.

media screen - ההגדרה landcape רוחב גדול מהגובה.

portrait - גובה גדול מרוחב.

-visibility: אם זה hidden זה תופס מקום ומסתיר.

diplay:none - אין אלמנט בכלל (כלומר לא תופס מקום).

display: block חוזר להציג אותו.

vh,vw זה באחוזים ביחס ליחס גובה/רוחב, למשל 1vh זה אחוז אחד גובה מהמסך.

html{font-size:62.5%;} יתן ברירת מחדל של rem ביחס 1:10 של הפיקסלים.

סדרי חשיבות – קודם Inline אחרי זה style של ה-HTML ואחרי זה בקובץ CSS.

before ו after - בד"כ זה יהיה רלטיבי באלמנט, ושמה ב- before ואפטר נשים אבסולוטי

ונשים 0 במיקומים שלהם ונשלחק את זה (מיקום כמו left וכו').

Before - אם רוצים לפני או מאחורי האלמנט (כלומר בדף האחורי).

after -זה יהיה מעל האלמנט כמו דף שנמצא על דף אחר.

media screen - ההגדרה landcape רוחב גדול מהגובה.

portrait - גובה גדול מרוחב.

-visibility: אם זה hidden זה תופס מקום ומסתיר.

diplay:none - אין אלמנט בכלל (כלומר לא תופס מקום).

display: block חוזר להציג אותו.

**משתנים ב-CSS:**

אם באחל השורות נרשום למשל --light-color:value;

אז אם נרצה להשתמש בזה במקום אחר למשל בצבע רקע.

אז נרשום - background: var(--light-color)

יוצרים סוג של סלקטור בסטייל או בקובץ CSS שנקרא

לו :root{} ובפנים נכניס את כל המשתנים.

למה זה טוב?

אם למשל עכשיו שמנו בכמה מקומות גוון צבע מסוים במקום להחליף בכל המקומות

פשוט נשנה את ערך הצבע במשתנה וכל שאר המקומות יעודכנו (כי השתמשנו

בהם גם כן במשתנה).

אפשר לרשום כל שם משתנה העיקר שלפני השם שלו יהיה כתוב

--

אפשר לשים משתמש בכל מחלקה אבל זאת החלוקה הכי קלה במקום [

אחד מרוכז

**אנימציה:**

        animation-name: ex;//ניתן לבחירה

        animation-duration: 2s;//זמן מחזור בכל אנימציה

        animation-iteration-count: infinite;//כמות פעמים שחוזר על הפעולה

   animation-timing-function: ease-in-out;//כמו במצגת בהדרגה המהירות עולה בהתחלה

        animation: animate1 5s forwards ease-in-out;//קיצור בשורה אחת עושה כל מה שלמעלה

//

       transition-duration: 2s;

        transition:property : background;//יצור מעבר הדרגתי של 2 שניות בצבע הרקע

    }

    @keyframes ex {//נותן פרמטרים ספציפיים לאנמציה בשם שבחרנו

        to {

            width: 600px;//בעצם במשך 2 שניות במקרה הנ"ל יגיע להרחבה הדרגתית

של בוקס עד לרוחב כזה שרשמנו.

            background-color: red;//בהדרגה יגיע לצבע הזה

            top: 300px;//בהדרגה יגיע לגובה הזה

        }

    }

אם רושמים ב- keyframes משהו כזה:

25% {

Top:300px;

}

50% {

Top:600px

}

זה אומר אחרי רבע מהזמן יעשה את הבלוק הרשון, ברבע השני של הזמן את הבלוק השני וכך הלאה.

השורה זאת מגדירה אנימציה לבוקס ספציפית על הרקע ועל ה­border.

transition: background, border 2s ease-in-out;

ואם נרשום את המילה all באנימציה תהיה ממש על כל הרכיבים ברגע שנרחף מעל ה-בוקס.

transition: all 2s ease-in-out;

.box:hover{

background:red;

border-radius: 5px;

height:30px

}

----

background-color: rgb(159, 159, 216);

width: 30vw;

height: 30vh;

animation-name: ex;

animation-duration: 2s;

animation-iteration-count: infinite;

animation-fill-mode: forwards;

animation-timing-function: ease-in-out;//כמו במצגת בהדרגה המהירות עולה בהתחלה

animation: animate1 5s forwards ease-in-out;//קיצור בשורה אחת עושה כל מה שלמעלה

transition-duration: 2s;

transition:property : background;//יצור מעבר הדרגתי של 2 שניות בצבע הרקע

}

@keyframes ex {

from{ top:0;}

to {

width: 600px;

background-color: red;//בהדרגה יגיע לצבע הזה

top: 300px;//בהדרגה יגיע לגובה הזה

}

}

יצירת 3 דיבים מוכנים יחד עם שמות רציפים וגם שמות מחלקה

div.{Item$}\*3

אם רוצים ליצור אלמנט דיב עם שם מחלקה מוכן, עושים .class כאשר זה שם המחלקה שתהיה בשם של אלמנט הדיב ומיד יצור אלמנט שלם מוכן עם כיתוב המחלקה והכל בפנים מוכן ככה.:

    <div class="grid"></div>

**גריד**

מה עושים כדי להפעיל את גריִד?

יוצרים קונטיינר במכילים נניח אלמניטים (כנראה דיבים ואולי דברים אחרים)

אם נניח יש 3 כאלו בפנים אז במחלקה של הגריד יוגדר:

.grid

{

display: grid;

grid-template-columns:200px, 200px 100px;

grid-gap: 1rem;//מגדיר רווח מכל הצדדים של כל אלמנט בגריד

        grid-auto-rows: 2fr;//לשאר האלמנטים יקבע ערך 2 פַר במקום משהו ברירת מחדל

}

זה אומר שזה יסודר בשורה ואלה הגדלים של כל שורה (כלומר אם יש לנו 5 דיבים, אז זה יסודר לפי 3 דיבים בכל שורה בגדלים).

יש גם יחידת מידת בשם fr במקום היחידות האחרות. זאת יחידה מיוחדת גמישה, מה שאומר שאם נקטין את החלוק גם האלמנט יוקטן ולא יגלוש לשורה/עמודה הבאה.

לפעמים רוצים אלמנט, לא צריך להגדיר רוחב/גובה בהכרח, מספיק לתת לו padding וזה לבד ייתן צורה שלמה לאלמנט.

.header {

grid-area: header;

}

.content {

grid-area: content;

}

.sidebar {

grid-area: sidebar;

}

.box-1 {

grid-area: box-1;

}

.box-2 {

grid-area: box-2;

}

.box-3 {

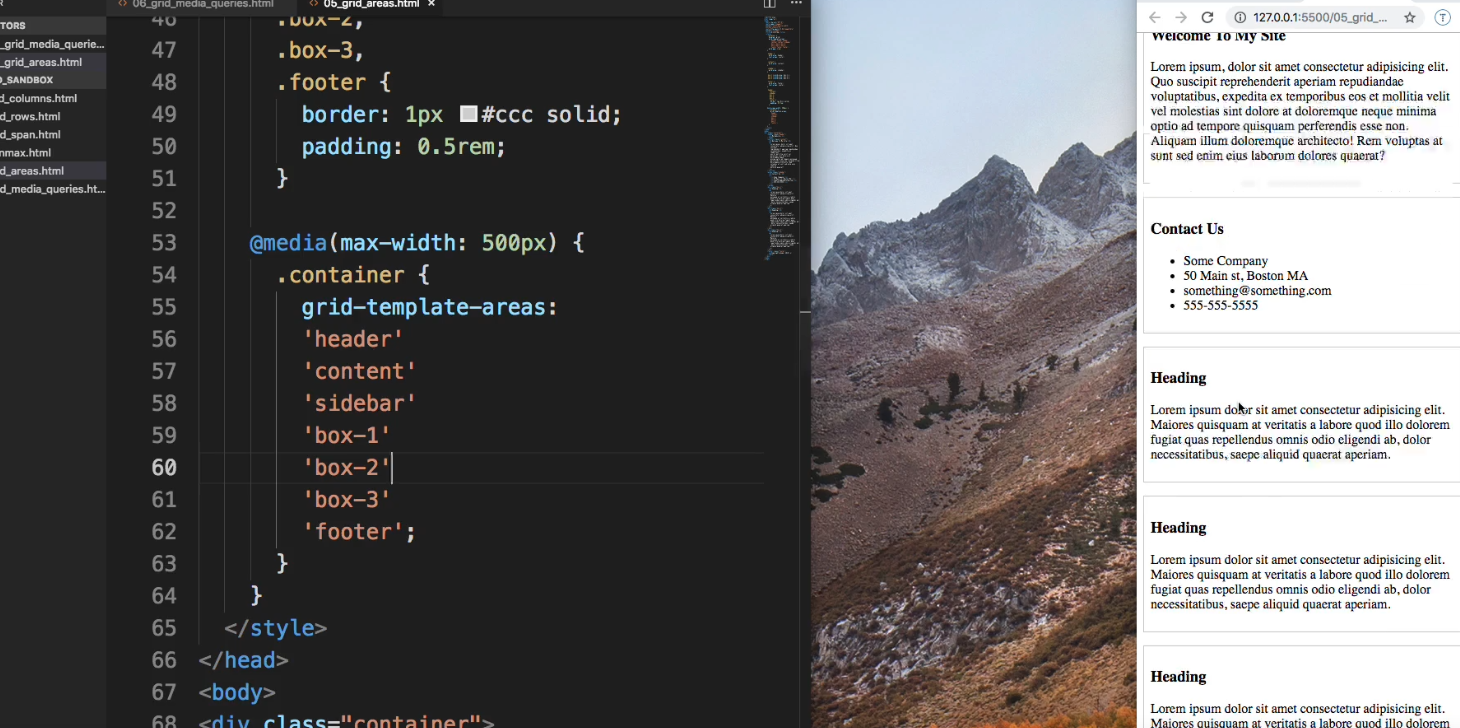
grid-area: box-3;

}

.footer {

grid-area: footer;

התאמה לסמטרפונים למשל:



דרך קיצור לרשום

grid-colum-start:1; grid-colum-end:4;

זה שקול ל- grid-coulm:1/span 3;

כלומר 1 אומר לו התחל מעמודה 1 ושמה תעשה החל ממנה 3 span ים.

חשוב ב-media screen חייב לשים את גדלי המסך מהגדול לקטן (כי אחרת זה יעצור למשל עבור טלפונים יציג רזולציה של טבלט אם בקוד של הטלפון מופיע לפני הטבלט. לכן עושים מהגדול ביותר ברזולציה לקטן ביותר.

הפקודה

grid-area: 2 / 1 / 2 / 4;

אומרת ברמת ה-item מגדירים אותה, ואומרת שיש לשים את הפריט בשורה 2 ובעמודה 1

ועד שורה 2 (2/1/2) ועד עמודה 4.

**רספונסיביות:**

מה ש-rem אומר זה שהגודל שלו הוא ביחס לדיפולט של הדפדפן

7:57

אם בד"כ הדיפולט הוא 16px אז 1rem = 16px קבוע

ואם אתה משנה את ההגדרות על אלמנט ה-html ככה

html {

/\* The default font size of the browser is 16px, so if we want 1rem to be equal to 10px, we divide 10px by 16px = 62.5% \*/

font-size: 62.5%;

}

אז 1rem = 10px

ואז זה מאפשר לך ב-@media להגדיר לו ככה

@media only screen and (max-width: 800px) {

html {

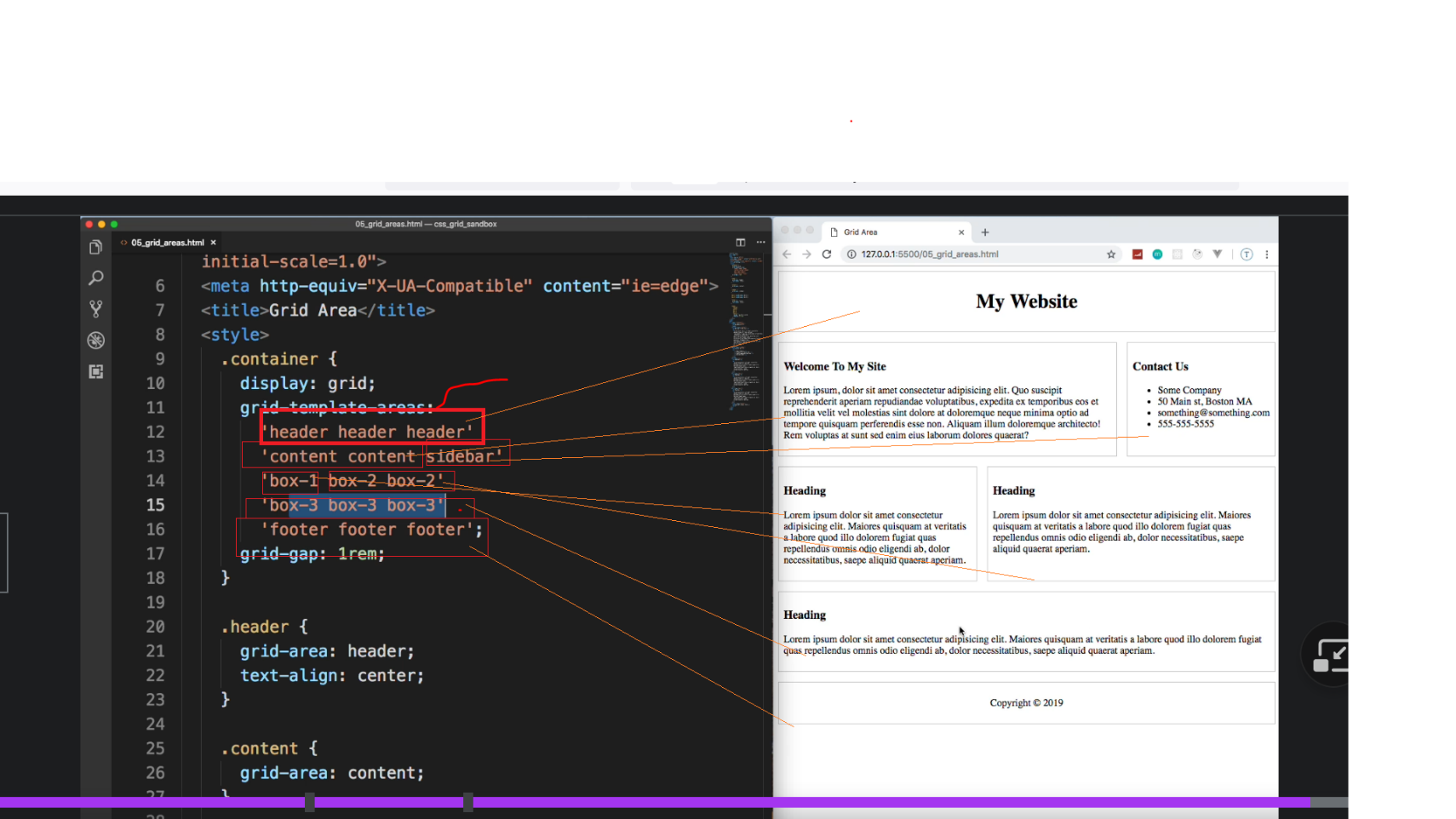
font-size: 50%;

}

}

אם כתבת הכל בפיקסלים ואתה רוצה לשנות את הגודל של כל האלמנטים בכמה אחוזים, תצטרך לעבור אלמנט אלמנט ולשנות ידנית

עם מה שעשינו עם rem אתה שולט בגודל במקום

JS

let- מגדיר משתנה, אבל המשתנה יהיה מוכר גם מחוץ לבלוק שהוגדר

var- המשתנה יהיה מוכר רק בבלוק שהוא הוגדר

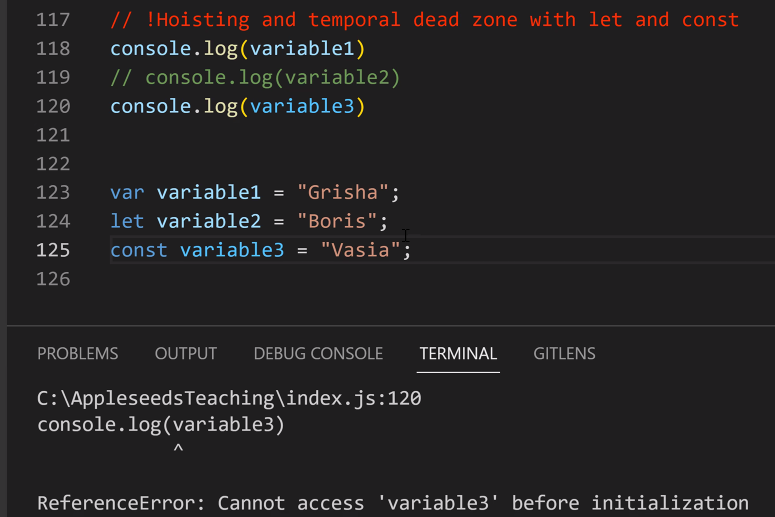
const - מגדיר קבוע

var דומה ל-let אבל מגדיר משתנה

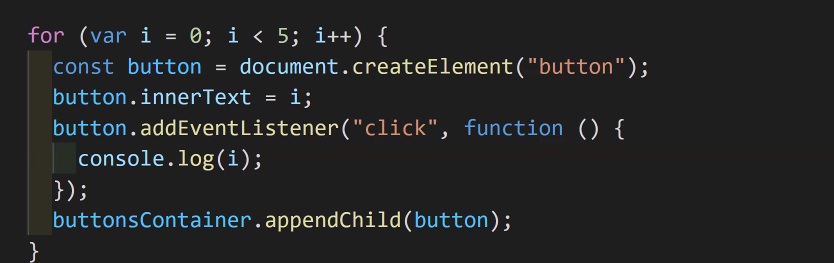
clear() מוחק את הקונסול

למעשה VAR שמוגדר בתוך פונקציה לא יהיה מוכר מחוץ לפונקציה.

אבל אם ה-VAR מוגדר בתוך בלוק של IF או FOR וכו', זה יהיה מוכר מחוץ לבלוקים של ה-IF או FOR בתוך אותו איזור אם זה פונקציה או קובץ ראשי, אבל אם למשל הגדרנו var בתוך if וזה בתוך פונקציה, זה לא יהיה מוכר עדיין מחוץ לפונקציה.



אבל דוגמא חשובה לגבי VAR:



בכל הפעמים יודפס i=5, כי ב-var ה-I יוצא מחוץ לפונקציה ולמעשה נשאר 5 בסיום ואם היה אירוע, זה הערך שיודפס.

מחרוזות:

.length יודע מה מספר התווים שלו

.trim - מוחקת רווח לבן מהמחרוזת

.indexOf - איפה נמצא ב-SUB STRING במחרוזת.

slice(index)- יחזיר את התת מחרוזות החל מאינדס ועד הסוף (מתחילים מ-0)

אפה גם לרשום טווח כמו (4,8)

replace(sub string, new string)- מחליף את סאב סטרינג בניו.

אם רוצים לרשום גרש ' בתוך מחרוזת, צריך לשים לפני את התו \.

אם רוצים גרשיים, צריך כנ"ל לעשות \ לפני כל גרש.

אם רוצים את התו \ צריך לרשום פעמיים ככה \\

T E M P L A T E L I T E R A L S

אם רושמים משהו כזה

let val='maor';

'game\_over ${val'}';

לשים לב זה גרש צדדי ולא רגיל (זה ידפיס את המחרוזת יחד עם ערך ה-val

-לא ניתן לרשום רק חישוב בתוך הדולר בלי מחרוזת לפני.

לשים לב ש- undefined יכול להיות במספר מקרים

למשל אם הכרזתי משתנה ולא שמתי בו כלום או שיש לי מערך ואני

מנסה לגשת לאינדקס שלא קיים בו.

לעומת זאת NULL זה ערך מפורש שהוקצה למשתנה

parseInt() - ממיר לשלם \_גם אם זה מעורב עם מילים בתנאי שיש מספר בהתחלה.

NaN - מציין שזה לא ערך מספרי (למשל אם עושים חלוקה ב-0 וכו' זה שונה מ-undeifned שזה בכלל לא מוגדר.

לולאת FOR EACH בJS

for (let color of arr)

בדיקת שוויון -JS : !==

בד"כ פונקציות הסטרגינים מחזירות ערך -1

במקרה שמשהו לא קיים וכו'.

זה מילון - { value: pick(values), suit: pick(suits) };

var- כבר לא בשימוש כי הוא מוכר בכל מקום ולא רק בבלוק.

JS נקרא הקוד מלמעלה למטה, וכנראה ב-CSS זה הפוך ונקרא למטה למעלה.

-שרוצים לאתחל ערך לא ידוע, לא נאתחל אף פעם ל- undeifned בגלל שה-js נותן ערך כזה, ואלו המכתנת רושם Null שהוא לא ידוע הערך.

console.log(y);//

console.log(x);//ייתכן undeifned \

בגלל שהמשתנה כן מוכר אבל כי זה ואר אבל עדיין לא הגיע לשורה של ההשמה

var x= 6;

const y =8;

הרו פונקשיין (נראה פונקציה אנונימית)

Const func () => a + b;

Const func ()=> {a +b };//זה מחזיר את הערך אחרי הסוגריים מסלולסלים רק אם הביטוי לא מורכב

Const function func() {}// כיתוב רגיל

// Function **declaration**

function greet(who) {

return `Hello, ${who}!`;

}

**// Function expression**

const greet = function(who) {

return `Hello, ${who}`;

}

**//arrow function syntax**

const greet = (who) => {

return `Hello, ${who}!`;

}

expr - זה בלוק של קוד

נניח הקוד שמחזיר משהו.

stactment - זה התיחסות שנייה.

const foo=()=>{};

זה נקרא הרו פנשקשיין.

אנחנו מעבירים לפונקציה ארגומנטים, ובהכרזה זה נקרא פרמטרים.

implict return

const func =num=>)num%2===0);

וזה זהה אבל בשורה בלי סוגריים

const func =num=>num%2===0;

יש אפשרו להשתמש ב-arror function פה למשל

const double =arr.map(n=>n\*2)

כאשר arr זה מערך, וזה יבצע את הפונקציה שהיא פרטמר

על כל ערך במערך.

וזה גם הערך המוחזר למעשה.

-אם רוצים פונקציה מצורה מפורשת (של הצהרה) לצורה של explicit and implicit הכוונה בכך היא:

const power = (a) => Math.pow(a, 2);

-ואם רוצים // From function expressions to IIFE functions.

זה אומר להפוך את זה :

const squareRoot = a => Math.sqrt(a);

לזה:

(function squareRoot(a) {Math.sqrt(a);})();

ואפשר גם לאחד משני הבאים:

(function (a) {Math.sqrt(a);})();

((a)=> {Math.sqrt(a);})();

כמה הבדלים בין Number ל- parseInt:

Number("")//0

parseInt("")//NaN

Number("123")//123

parseInt("123")//123

Number("123ac") //NaN,as it is a non numeric string

parsInt("123ac") //123,it parse decimal number outof string

Number(true)//1

parseInt(true) //NaN

parseInt("20px"); // 20

// type conversion

Number("20px"); // NaN

expr - זה בלוק של קוד

נניח הקוד שמחזיר משהו.

stactment - זה התיחסות שנייה.

const foo=()=>{};

זה נקרא הרו פנשקשיין.

אנחנו מעבירים לפונקציה ארגומנטים, ובהכרזה זה נקרא פרמטרים.

implict return

const func =num=>)num%2===0);

וזה זהה אבל בשורה בלי סוגריים

const func =num=>num%2===0;

יש אפשרו להשתמש ב-arror function פה למשל

const double =arr.map(n=>n\*2)

כאשר arr זה מערך, וזה יבצע את הפונקציה שהיא פרטמר

על כל ערך במערך.

וזה גם הערך המוחזר למעשה.

התנאי המקוצר של a>b? one:two

זה נקרא טרנרי.

אם רוצים להפעיל פונקציה בתנאי אפשר לעשות דבר כזה:

flag && callfunc();

זה יקרא לפונקציה רק את FLAG הוא true.

אם אחרי הערה // עושים ! זה מראה את זה בצבע אדום .

**מערכים:**

Push - add to end

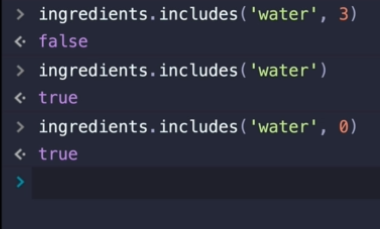
Pop - remove from end

Shift - remove from start

Unshift - add to start

concat - merge arrays

includes - look for a value



indexOf - just like str.indexOf

join - creates a string from arr

reverse - reverses an array!

slice - copy portion of an arr

splice - remove/replace elements

sort - sorts an array

    copyArrPeople.splice(2, 0, "Elizabeth", "Artie");

זה בעצם אומר לך לאינדקס 2, ומחוק 0 איברים משמה, וכל האיברים שאני שם שם, הוסף אותם החל מאינקס זה למערך.

הערה- פונקצית join למעשה מאפשרת לחבר תת מחרוזת למערך ולעשות הפרדה בין כל איבר במערך עם תת המחרוזת למשל

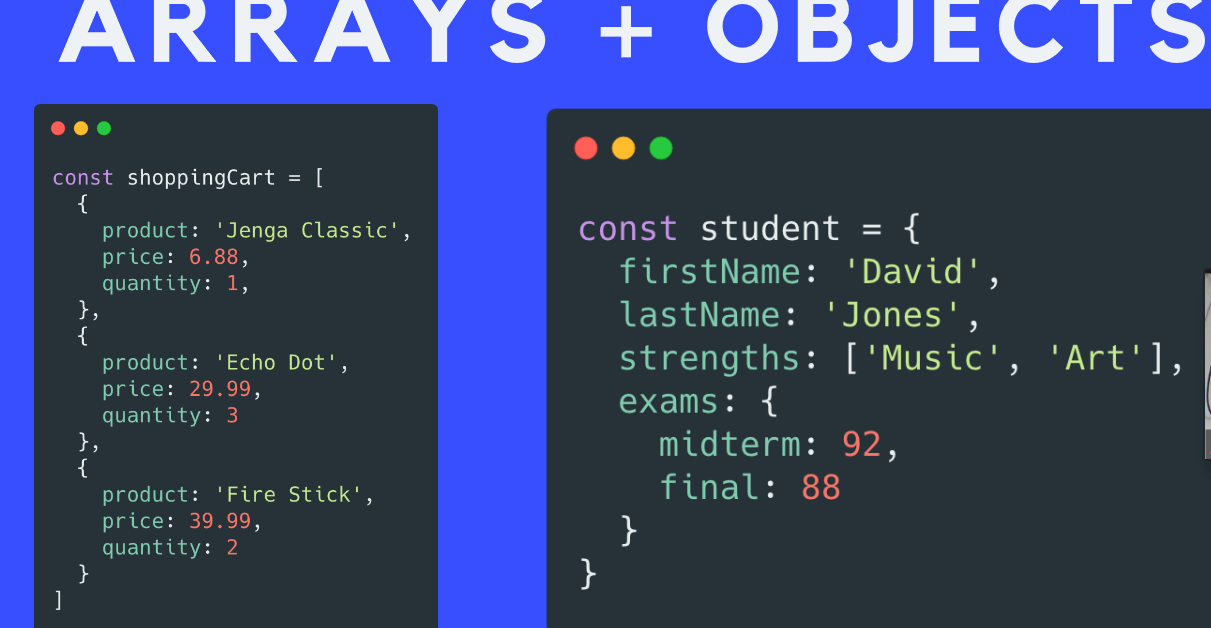
word.toUpperCase().split('').join('.')

לוקחת את המערך word הופכת אותו לאותיות גדולות ועושה הפרדה בין רווחים כאשר split מחזירה מערך חדש, ואז join יוצרת הפרדה של . בין כל איבר במערך כך שהפלט נראה כך ביחס לקלט:

//["A.S.A.P", "B.Y.O.B", "R.S.V.P", "D.I.Y"]

Objects are collections

of properties.



אם עושים let [a,b] = func();

כאשר func מחזירה return [a,b] זה פשוט מחזיר כמה ערכים

וכל אחד הולך למשתנה אחר.

הפקודה arr2=[1,...arr,4] שופכת את תוכן המערך לתוך מערך אחר.

המילה Prototype שמופיעה בפונקציות של JS זה הכוונה שזה פועל על הנגזרת

ובלי זה, זה פועל על אלמנט האב.

מערכים ב-JS הולכים לפי רפרנס (וכך גם כל האובייקטים), לחץ אם עושים arr2=arr אז כאשר arr משתנה גם arr2 וזה לעומת משתנים רגילים פרימיטביים ששם זה לפי ערך ולא לפי רפרנס.

יש לשים לב- שהערכים שבתוך האובייקט עצמו, כן לפי value וכן אפשר לשנות ערך שם.

json- מבקשים מהאתר מסוים ומקבלים מערך של אובייקטים

עפ מידע די דומה ל-DB.



shallow copy -יוצר העתק של אובייקט אבל עדיין מצביע לאובייקט המקורי,ולכן אם האובייקט המקורי השתנהגם האובייקט החדש ישתנה לערך זהה.

deep copy - למעשה יוצר העתק אחד לאחד עם כתובת שונה וחדשה וזכרון.

הכרחי כשיש אובייקט או מערך שיש בתוכו עוד אובייקטים או מערכים, יעני אובייקט\מערך מכונס.

בשביל ליצור עותק שהאובייקטים\מערכים המכונסים לא מושפעים מההעתק צריך לעשות העתקה עמוקה

איך מחלצים מידע כפי שרוצים ממערך למשתנים:

arr[1,2,3,4,5,6]

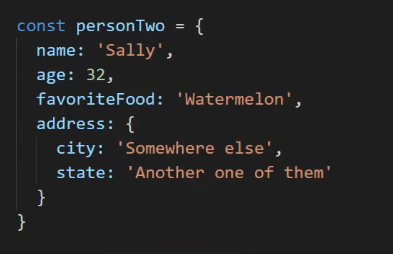
arr2=[a,b,,,c,...arr3]

ייתן ערך של 1 ו- 2 ל- a,c בהתאמה. על 5 ו-6 יידלג, ואת השאר ישים במערך בשם arr3 החלק של ...arr3 תמיד יהיה בסוף וזה נקרא rest

פירוק סטרינג לתוך מערך אותיות ע"י str=[...arr]

בתוך `s${[...a]}` ככה שופכים מערך, כלומר חייב לשים סוגריים מרובעות.

אם יש לנו למשל:



const {name: "yosi",age,…rest} = personTwo

למעשה זה מגדיר name ברירת מחדל בהנחה ואין שדה כזה, ואם יש שדה כזה, זה יחזיר את ערך אותו שדה.

עכשיו בנוסף שמנו age, אז זה יחזיר את ערכו, ו-…rest מחזיר לנו את שאר הערכים שלא ציינו במפורש ושם בתוך האובייקט בתוך מערך שנקרא rest (אפשר לציין איזה שם שרוצים).

אם נרצה ערך של שדה של אובייקט בתוך אובייקט כמו city נרשום:

const {address {city}} = personTwo

בצורה דומה נוכל להעביר אובייקט לפונקציה ולבחור לקבל ישירות את השדות שרלוונטים לנו ולעשות בהם שימוש כמשתנים עצמאים ללא שימוש באובייקט, כך:

Function f({name, age}){

Clg(name, age)}

f(personTwo)

באובייקטים שנרצה לחלץ, שם המשתנה נחלץ לפי שם השדה באובייקט, ולכן אם יש כבר משתנה בקוד בשם זה, נרשום בצורה כזאת name:name2 וכו'

const {name:name2, age:age2} =  data[2];

console.log("---"+name2)

const {city:city2} =  data[2].address;

console.log("---"+city2)

יש אפשרות ליצור בתוך אובייקט משתנה, כאשר שם המשתנה יהיה למעשה מחרוזת כמו בדוגמא הבאה:



איך מגדירים בתוך אובייקט פונקציות?

const obj = }

divide : function (x, y)

{}, squre :function (x) {}

};

//call function

math.divide(5, 9);

זה למעשה החלון:

console.log(this)

this- איך להשתמש כדי לעשות השמה של ערכים לתוך משתני האובייקט:

const person = {

first: 'Cherilyn',

last: 'Sarkisian',

nickName: 'Cher',

fullName() {

//In a method, this refers to the object the method "lives" in:

const {

first,

last,

nickName

} = this;

הערה- אי אפשר לעשות מעבר על איברי האובייקט בצורה של for( let x: obj) לא אפשרי!

זאת כן הדרך לקבל נתוני האובייקט:

for (let **movie** of **Object.keys**(movieReviews))

מה שבצבע צהוב – זה אובייקט, עכשיו הפונקציה object.keys לוקחת את כל המפתחות (שהמפתחות זה השדות שיש באובייקט (ולא הערכים שלהם).

לעומת זאת

movieReviews[movie]

זה ממש הערך שמתאים לכל מפתח.

וזאת דרך נוספת לקחת את הערכים מתוך האובייקט:

for (let r of Object.values(movieReviews))

בעצם באובייקטים תמיד ידברו על key, value כאשר זה נקרא גם אוסף של מאפיינים ואין גישה של אינדקים, אלה גישה לפי key כדי להשיג את ערך (value) המאפיין.

יש לשים לב, אם יש לנו מערך ובו יש אובייקטים, כלומר:

arr[{},{}] אז אם רוצים לגשת לשדה שלי אחד האובייקטים, חייב קודם לגשת לאובייקט

עצמו ע"י [i] ולא ניתן ישר לעשות arr.name למשל בלי לעשות arr[i].name

כי השדות לא נראות ברמה החיצונית עד שלא ניגשים תחילה לרמת האובייקט.

אלה גישות זהות למפתחות באובייקט:

palette.red; //'#eb4d4b'

palette['red']; //'#f9ca24'

יש לנו את האובייקט הבאה, וזאת הדרך גישה למידע בו:

 const id = pages['/users'][0].id;//מה שנקבל זה את ה-1 שסימנתי למטה

לשים לב- כל אובייקט שבתוך אובייקט אחר, ניגשים בצורה הבאה:

Pages – הוא אובייקט .

'/users' – זה שדה בתוך אובייקט, ואליו ניגש ע"י [].

בגלל שאנחנו רוצים את האובייקט הראשון שבתוך '/users', נרשום מיד אחרי בלי שום דבר מפריד עוד [] שבתוכם מיקום האובייקט המבושש אחנו רוצים.

עכשיו התוצאה היא אובייקט חדש, ואנחנו רוצים את השדה id, אז ניגש אליו ע"י .id אבל אפשר גם על ['id'] כלומר אפשר היה לרשום pages['/users'][0]['id']

גדגכג

            const pages = {

                '/users'        : [

                    { id: 1, username: 'Bilbo' },

                    { id: 5, username: 'Esmerelda' }

                ],

                '/users/1'      : {

                    id        : 1,

                    username  : 'Bilbo',

                    upvotes   : 360,

                    city      : 'Lisbon',

                    topPostId : 454321

                },

                '/users/5'      : {

                    id       : 5,

                    username : 'Esmerelda',

                    upvotes  : 571,

                    city     : 'Honolulu'

                },

                '/posts/454321' : {

                    id    : 454321,

                    title :

                        'Ladies & Gentlemen, may I introduce my pet pig, Hamlet'

                },

                '/about'        : 'This is the about page!'

והנה דוגמא מסובכת כוללת השמה למקום מורכב עם מערכים ואובייקטים מקוננים:

school['teachers'][1].subjects[1]='a'

const school = {

  teachers: [

    {

      id: 1,

      name: "Pinchas",

      subjects: ["chemistry", "biology", "physics"],

      students: [],

      capacityLeft: 3,

    },

    {

      id: 2,

      name: "Williams",

      subjects: ["history", "ethics"],//פה עשינו השמה של איי

      students: [],

      capacityLeft: 2,

    },

  ],

  students: [

    {

      id: 10,

      name: "Jennifer",

      age: 20,

    },

    {

      id: 11,

      name: "Howard",

      age: 23,

    },

    {

      id: 12,

      name: "Old-Timmy",

      age: 86,

    },

    {

      id: 13,

      name: "Houston",

      age: 21,

    },

  ],

};

console.log(school['teachers'][1].subjects[1]='j')

hoisting - שימוש במשתנה לפני שהוא מוכרז בקוד

כמו לבדוק ערך שבו למרות שבכלל לא אותחל וכו'.

ההוסטינג רק בהצהרה ולא על האתחול ב-JS.

ב-JavaScript, למשתנה לא מוכרז מוקצה הערך לא מוגדר בביצוע והוא גם מסוג undefined.

SHORTHAND - כנראה שזה קריאה למתודה בצורה סטטית מבלי ליצור מופע לפני.

ב-js אפשר לגשת למשתנה ישירות בתוך obj (מופע של אובייקט) מבלי בעיות של הרשאה.

דומה להרשאת Public משפות עיליות. למשל obj.name="maor"; אפשרי בלי בעיה לבצע.

מציאת מקס' במערך -

math.max(...arr).

toFixed(1)- לוקח ספרה אחת אחרי הנקודה.

גם באובייקטים אפשר לעשות פריסה ע"י ...obj.

בנוסף כאשר יש לנו אוביקטים עם עם ערך נניח color:'color'

אז אם רוצים להעביר את הערך של color למעשה בגלל שגם

המפתח וגם הערך זהים, מספיק לרשום בהעברה בפונציה את המילה

color, ואם הערך לא זהה למפתח, אז צריך לרשום הכל.

הדבר הבא זה אובייקט (ממש דומה למחלקה של פרסון):

function Person (name, age){

this.name = name;

this.age = age;

}

const p = Person('maor', 18);

**פונקציות כארגונט ו-CALLBACK FUNCTION**

//We can store functions in an array!

const operations = [add, subtract, multiply, divide];

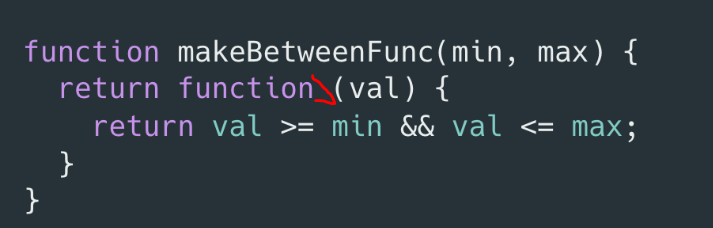
// We can also store functions in objects!

const thing = {

doSomething: multiply

}

thing.doSomething(4, 5) //20



btn.addEventListener('click', function () }

alert("WHY DID YOU CLICK ME!!??")

})

setTimeout(function () **{** //we pass an anonymous callback function

alert("WELCOME!");

**}**, 5000);//ה5000 זה הפרטמר השני של טייםאאוט

**ההבדל בין לולאת for in ולולאת for of**

let list = [4, 5, 6];

for (let i in list) {

console.log(i); // "0", "1", "2",//בעצם מחזיר אינדקים

}

for (let i of list) {

console.log(i); // "4", "5", "6"

}

קורא פעם אחת בלבד לפונקציה שנגדיר.

const updateTime = () => {

timer.innerText = `${d.getHours()}:${d.getMinutes()}:${d.getSeconds()}`;

};

stopTimeout = setTimeout(updateTime, 1000);

clearTimeout(stopTimeout);//מבטל את השעון

קורא כל כמה זמן קבוע לפונקציה

////////////////////////////////////////

const updateTime = () => {

d = new Date();

timer.innerText = `${d.getHours()}:${d.getMinutes()}:${d.getSeconds()}`;

clearInterval(stopInterval);//ביטול

};

stopInterval = setInterval(updateTime, 1000);

**המשך אובייקטים**

סביבת הרצה של strict mode במקרים שאין זה קוד יכול להיכשל ולא יהיה חיווי

עם שגיאה מ-JS ואילו עם המוד הזה, עוזר להימנע משגיאות לא ידיעות בזמן הרצה ופיתוח ב-JS.

אובייקטים זה המבנה הנתונים הכי נפוץ ב-JS.

ה-DATA עובר בצורת JSON בשרתים.

זה צמדים של KEY ו- VALUE.

שרוצים להוסיף שדה חדש לאובייקט והשדה לא קיים, פשוט עושים:

obj.name = 'f'; כאשר name זה השדה החדש שהוספנו.

delete obj.name מוחק שדה מהאובייקט.

באופן דומה אפשר למחוק את כל האובייקט.

הערה – לא נגדיר בתוך אובייקט פונקציה בצורה של => כי יוצר איזה בעיה שאמר, ולכן תמיד נרשום function.

חשוב אם מעבירים כפרמטר מחרוזת ורוצים לגשת לשדה הזה יש לרשום כך:

const findPerson =(type, id)=>{

    let retPerson ;

    school[type].forEach(element => {

    if(element.id == id) retPerson = element;

  });

  return retPerson;

}

console.log(findPerson('teachers', 2));

באמצעות הפונקציה אפשר לדעת האם יש שדה כזה בתוך האובייקט.

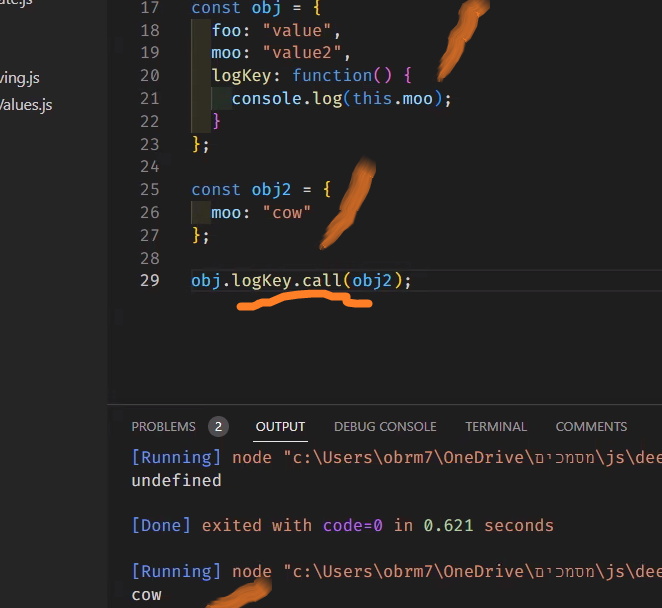
const example = {};

example.hasOwnProperty('prop'); // false

example.prop = 'exists';

example.hasOwnProperty('prop'); // true - 'prop' has been defined

עם השיטה call() אתה יכול לכתוב שיטה שניתן להשתמש בה על אובייקטים שונים.



בעצם forEach(item פועלת בצורה כזאת על כל חלק שנמצא ב-data שזה אובייקט.

.

חשוב אין דבר כזה ביטוי של ret = bbb.foreach(…return val)

ולכן חיי להשים בפנים את ההשמה כי איזה דבר כזה לעשות השמנה מחוץ לפקודה של foreach.

הפונקיה foreach עובדת על מערכים, ולא על אובייקטים וכו'

לשים לב שלפעמים אני מתבלבל וחושב שאובייקט זה מערך, פשוט אם רואים סוגריים מרובעות זה מערך וסוגריים מסולסלות זה אובייקט תמיד גם בכמויות גדולות של מידע.

  data[key].forEach(item => {

פונקציה זאת לא פועלת על STRING.

דרכים שונות לעבוד עם FOREACH:

const numbers = [20, 21, 22, 23, 24, 25, 26, 27];

// Using anonymous function expression:

numbers.forEach(function (num) {

console.log(num \* 2);

})

function printTriple(n) {

console.log(n \* 3);

}

// Using a pre-defined function:

numbers.forEach(printTriple);

// Using the index:

numbers.forEach(function (num, idx) {

console.log(idx, num);

});

הפקודה Object.entries(object1) למעשה מחזירה צמד מערכים של KEY,value של האובייקט הנתון שהעברנו בפרמטר.

הפקודה Object.assign(target, source) עושה העתקה של שדות (יחד עם ה-value שלהם) מהאובייקט src לאובייקט trg.

כלומר כל שדה שאין ב-trg אז יועתק יחד עם הערך שלו לתוך trg.

Object.create is a shallow copy only

לכן אם משנה שדה בעותק החדש

גם בקודם זה ישנה לאותו ערך.

פנקצית MAP:

const numbers = [1, 2, 3, 4];

const filteredNumbers = numbers.map((num, index) => {

if (index < 3) {

return num;//זה אפשרי לא חייב לרשום דבר כזה במקרה זה כן מימשו כך

}

});

// index goes from 0, so the filterNumbers are 1,2,3 and undefined.

// filteredNumbers is [1, 2, 3, undefined]//פלט

// numbers is still [1, 2, 3, 4]

יש הרבה סוגים של קריאות ל-MAP כרגע מרשום מה רלוונטי:

// Arrow function

map((element) => { /\* … \*/ })//כל אלמנט במערף

map((element, index) => { /\* … \*/ })//כנל אבל יחד עם אינדקים

map((element, index, array) => { /\* … \*/ })כנ"ל יחד עם מערך?

לוודא – אבל נראה ש-MAP רץ על על מערך ולא על סטרינג.

Array:

מתודות שיש במבנה הנתונים הזה:

const found = array1.find(element => element > 10);

מחזירה את האיבר הראשון שהיא מוצאת במערך שעונה על התנאי שמצוין בפנים.

const array1 = [1, 2, 3];

console.log(array1.includes(2));

// Expected output: true

בודקת שיש איבר כזה במערך.

const movie2 = movies.find(m => m.indexOf("Mrs") === 0);

מחזירה את האינדס הראשון במערך ששם המילה mrs היא מתחילת המערך (כלומר עם אינדקס 0).

המתודה filter:

const words = ['spray', 'limit', 'elite', 'exuberant', 'destruction', 'present'];

const result = words.**filter**(word => word.length > 6);

console.log(result);

// Expected output: Array ["exuberant", "destruction", "present"]

היא עושה פילטור למערך לפי תנאי מסויים, ומחזירה את כל האיברים במערך שמקיימים את התנאי הזה.

אפשר גם לעשות שאילתה קצת יותר מורכבת:

// Simple search functionality:

const query = 'The';

const results = books.filter(book => {

  const title = book.title.toLowerCase();

  return title.includes(query.toLowerCase())

})

הערה חשובה מאוד:

פונקציית המפה **משמשת למיפוי ערך אחד לאחר,** אבל שרוצים לעשות סינון איברים במערך עושים שימוש בפונקצית פילטר.

המתודה every רלוונטית למערכים (אפשר גם באובייקטים ע"י שימוש ב- prototype.every.

words.every(word => word.length === 3);

בודקת את כל האיברים במערך שמקיימים את התנאי הנתון ומחזירה true אם כן.

הפונקציה some עושה דבר זהה רק בודקת אם יש לפחות איבר אחד שמקיים את התנאי ואם כן מחזירה true.

פונקציות:

פונקציה טהורה בכל קריאה תחזיר לי אותו ערך.

פונקציה לא טהורה - בקריאה אחת ערך אחת מחזירה בקריאה אחרת ערך אחר

למשל בגלל שיש משתנה חיצוני שהשתנה והיא משתמשת בו (כמו משתנה גלובלי).

מבני נתונים שונים:

אם עושים const m = new map()

ואז (למשל) m.set('a',5); הוא פשוט מאפשר להוסיף ככה

מפתח וערכים למשתנה.

-הפונקציה m.open('a') יחזיר לנו את הvalue שמותאים למפתח.

זה פשוט מבנה נתונים MAP.

m.delete() מוחק שדה שרוצים

אפשר לממש עם זה מילון או גרפים.

לשים לב ב-MAP זה מימוש עם STRING-ים למשל אפשר לעשות m.set('full name',maor')

דבר שאי אפשר לעשות ב-אובייקטים, כי שם יש רק מילה אחת שאפשר להכניס כמו משתנה

ולא שני מילים (כלומר אי אפשר באובייקטים רווחים אבל ב-MAP כן).

פונקצית איטרטור -

let it - m.entries();

let next =

מבנה נתונים SET:

כל נתון יכול להיות רק פעם אחת (אם הכנסנו יותר הוא לא ייתחס אליו)

const s = new Set(['pasta', 'pizza', 'pizza'])

s.has('pizza') //-> true

for ( let val of s) //מעבר על איברי הסטים

פונקציות:

פונקציה טהורה בכל קריאה תחזיר לי אותו ערך.

פונקציה לא טהורה - בקריאה אחת ערך אחת מחזירה בקריאה אחרת ערך אחר

למשל בגלל שיש משתנה חיצוני שהשתנה והיא משתמשת בו (כמו משתנה גלובלי).

פונקצית sort

יש לזכור sortמשנה את המערך המקורי.

לכן נשתמש ב-slice כי אחרת זה ימיין את המערך 3 פעמים.

או בגלל שהיא יותר בזבזנית כנראה בסיבוכיות מקום, ואפשר להשתמש או ב-slice או ב-2 מערכים זהים (אפשר גם לעשות prices.sort אבל זה עניין הסיבוכיות מקום שעולה בה).

נעשה דבר כזה כדי למיין:

prices.slice().sort();

sort תמיין רגיל (לפעמים אפילו בלי הגיון אם למשל יש מספרים אחרי הנקודה, ולכן נשתמש בהדגשות הבאות בקוד), אם רוצים בסדר עולה או יורד ספציפית, נעשה דבר כזה:

// Ascending Sort:

const ascSort = prices.slice().sort((a, b) => a - b);//סדר עולה

// Descending Sort:

const descSort = prices.slice().sort((a, b) => b - a);//סדר יורד

איך עושים מיון לפי איבר מסוים?

// Sorting books by their rating:

books.sort((a, b) => b.rating - a.rating)

books זה מערך עם הרבה דשות, בחרנו שדה מסוים (דירוג במקרה זה) והסדר הוא סדר עולה.

השיטה **reduce**() מבצעת פונקציית callback של "מפחית" שסופק על ידי המשתמש בכל אלמנט של המערך, על מנת להעביר את ערך ההחזרה מהחישוב באלמנט הקודם. התוצאה הסופית של הפעלת המפחית על פני כל האלמנטים של המערך היא ערך בודד.

בפעם הראשונה שהקריאה חוזרת מופעלת אין "ערך החזרה של החישוב הקודם". אם מסופק, ניתן להשתמש בערך התחלתי במקומו. אחרת אלמנט המערך באינדקס 0 משמש כערך ההתחלתי והאיטרציה מתחילה מהאלמנט הבא (אינדקס 1 במקום אינדקס 0).

למעשה ה-total מקבל בהתחלה את הערך של אינדקס 0 וה-curr של אינדקס 1, ובאיטרציה הפנימית הבאה זה כבר יהיה total הסכום מפעם קודמת, ואילו ה-curr יהיה אינדקס 2.

אבל שעובדים עם ערך התחלתי ושמים ערך נניח 0, אז זה יהיה הערך שלו.

const nums = [3, 4, 5, 6, 7];

// To multiply all values in nums:

const product = nums.reduce((total, currentVal) => {

return total \* currentVal;

});

// total currentVal returnVal

// 3 4 12

// 12 5 60

// 60 6 360

// 360 7 2520

//Final Return Value: 2520

יש גם אפשרות לעשות reduce עם ערך התחלתי שהוא יוסיף סכום בסך הכל:

const total = [10, 20, 30, 40, 50].reduce((sum, currVal) => {

return sum + currVal;

}, **1000**) //sum starts at 1000,

הערה- באופן זהה הערך ההתחלתי יכול להיות אובייקט (כלומר לרשום במקום 1000 את המספר {}) ואז זה מלביש בעצם את התוצאה מעל האובייקט נניח שדות חדשים וכו' כמו בדוגמא הבאה):

const results = votes.reduce((tally, val) => {

tally[val] = (tally[val] || 0) + 1;

return tally;

}, {});

קיצור דרך במקום תנאי ע"י שימוש בתנאי ||

tally[val] = (tally[val] || 0) + 1;

זה למעשה יוסיף 1 לאיבר אם הוא true ואחרת ישים בו את הערך 1.

אפשר לעשות דבר כזה בתוך reduce

    let sum = 0;

    max = arr.reduce((a, b) => {  sum+=b;  return sum/arr.length}, 0);

פונקציות וסקופים ודיבאג:

let animals = ['grizzly bear', 'panda bear', 'spectacled bear'];

var i = 10;

for (var i = 0; i < animals.length; i++) {

  console.log(i, animals[i])

}

console.log(i)//הפלט יהיה 3 כיוון שברגע שהכרזנו מחדש על המשתנה אַיי מיד ההכרזה הקודמת בוטלה למעשה

let animals = ['grizzly bear', 'panda bear', 'spectacled bear'];

let i = 10;

for (let i = 0; i < animals.length; i++) {

  console.log(i, animals[i])

}

console.log(i) //כאן הפלט יהיה 10 כי בלולאה זה למעשה משתנה אַיי חדש שמוכר רק בתוך הבלוק של הלולאה וכאשר יוצאים מהלולאה נגמר הבלוק ושם הוא מכיר את האַיי הקודם

a אוטומטית גלובלי אם רשמנו a=20 בתוך פונקציה ויהיה מוכר בכל מקום

גם מחוץ לפונקציה

אבל אם רשמנו var a בתוך הפונקציה, זה יהיה מוכר רק בתוך הפונקציה

כי זה עדיין לא גלובלי VAR, זה מוכר בכל אותו הבלוק שהוא שייך אליו.

אם רוצים לקרוא לפונקציה שהגדרנו ממש את הפונקציה עצמה כ-VAR או כ-LET

אי אפשר לקרוא לה לפני שהצהרנו עליה.

אבל אם רושמים בצורה של דקלריישן שזה funtion (){} אז כן אפשר יהיה

לקרוא לפונקציה לפני הצהרת הפונקציה.

בעצם הצורה היחידה שבה אפשר לעשות Hostion לפונקציה זה רק ע"י הכיתוב

של function.

function scope - זה האיזור הסקופ של הפונקציה.

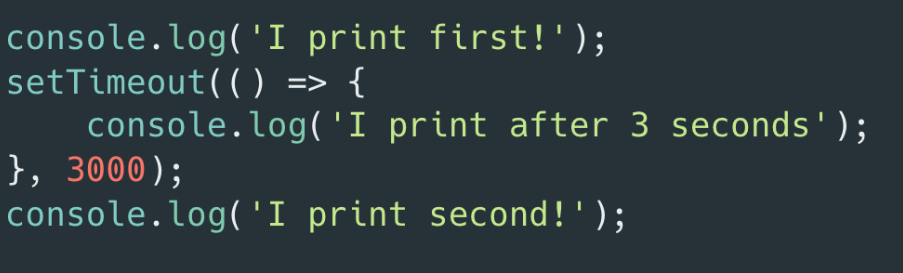
bloack scope - זה כל האיזור הקוד של הבלוק

lexical scope - נראה שזה פונקציה בתוך פונקציה והאיזורים המוכרים.

\

single JS thread

בכל נקודת זמן נתונה, זה חוט JS יחיד פועל ב שורה אחת של קוד JS



במצב שיש לנו שתי קריאות זהות עם פונציה בשם זהה בקוד (עם המילה השמורה function)

הקריאה שתיקרא תהיה האחרונה בלבד בקוד.

דוגמא לקטע קוד ומשהו ייחודי:

for (i = 0; i < 3; i++) {

    const log = () => {

    console.log(i);

    }

    setTimeout(log, 100);

}//for

הקטע קוד הבא ידפיס 3 פעמים 3, ולמה?

כי בלי הקטע של setTimeout למעשה log לא תיקרא לעולם, זה פשוט הגדרה של פונקציה בתוך הבלוק של הלולאה.

עכשיו בקריאה של setTimeout בכל איטרציה יש לנו קריאה אחת שנשמרת במחסנית ותיקרא מאוחר יותר. בגלל שעברו 100 מילי שניות, קטע הקוד של הלולאה כבר הסתיים, והערך של i כבר הגיע לערך המקסימלי שלו שזה 3.

עכשיו אחרי 100 מילי שניות מתבצע הקריאות של Log שזה 3 רצופות, ובזמן הזה כבר כולם עם הערך 3, ולכן זה הפלט .

דוגמא לפונקציה ומשתנה בשם זהה ומה זה אומר ב-JS:

var a = 1;

function b2() {

  a = 10;

  return;

  function a() {}

}

b2();

console.log(a);

בגלל שהוגדרה פונקציה בשם a, אז js יוצרת באופן אוטומטי תמיד גם משתנה בשם זהה לפונקציה, לפיכך יש גם בזיכרון בתוך הסקופ הזה משתנה בשם a.

אבל לא נראה לי שזה משנה, כי המשתנה a שיש אליו השמה בתוך הפונקציה זה למעשה משתנה גלובלי ולא נראה שזה בכלל קשור למשתנה חיצוני var a.

מכאן כאשר קראנו לפונקציה b2, ההשמה של a=10 נעשתה למעשה ל-a הקרוב ביותר, במקרה הזה זה ה-a שנמצא בתוך הסקופ שזה בדיוק הפונקציה a ולכן ערכו השתנה ל-10.

closuer

closures- כאשר פונקציה יכולה לגשת לscope של האבא

אפילו אחרי שהאבא נגמה.

למשל יש לנו פונקציה עם משתנה מקומי ובתוכנה יש פונקציה פנימית.

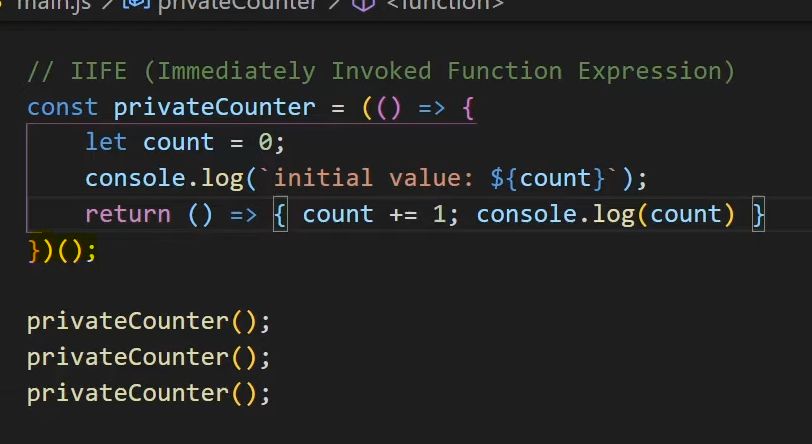
עכשיו הפונקציה החיצונית מחזירה כ-return את הפוקנציה הפנימית.

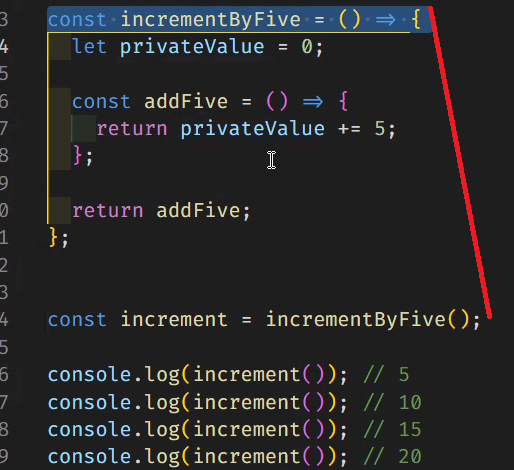
מה שקורה המשתנים המקומים שהיו ואותחלו במקור בפונקצית האב, יהיו זמינים

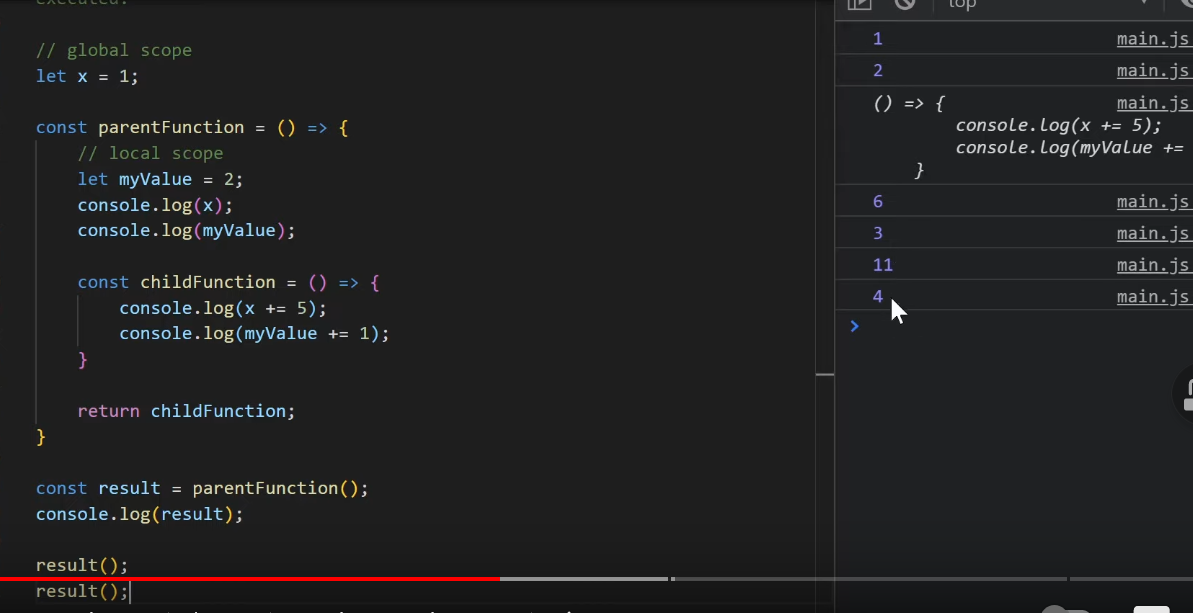
עכשיו בקריאה בפוינטר לפונקציה שקיבלנו בחזרה, כי למעשה הפונקציה הפנימית

התקבלה עם כל תמונת המשתנים שהיו בסביבה שלה.

באופן כללי var, let לא מאותחלים לערך ברירת מחדל ואילו var מאותחל ל-undefined.







דוגמאות נוספות:

closuer-

const makeAdder =()=>{return (n)=>{return num+n}};

const addFive = makeAdder(5);

console.log(addFive(4));

ידפיס 9, בעצם קישרנו את ה-5 למשתנה num ובקריאות הבאות הוא מחובר ונשאר להוסיף רק את n.

הפעלת דיבאג של ה-dev של כורום.

רושמים את המילה debugger תוך הקובץ איפה שהוא, ואז שמפעילים את הקובץ בדפדפן בתוך לשונית console זה מאפשר להגדיר מקומות של Breack point ולבדוק.

אם מוסיפים בתוך ה-conlose הדפסות עם עוד איזה סימון נגיד סמיילי או משהו.

ואז אפשר לחפש בתוך הקונסול סמיילי ויציג כל המקומות שרלוונטים עם הקונסול.

אמר היחס של כתיבת קוד ובדיקת דיבאג, זה יחס של 1 ל-5

בעצם 20 אחוז מהזמן כתיבת קוד והשאר דיבוג.

|  |  |  |
| --- | --- | --- |
| שם הפונקציה | מה עושה | איך עובדת |
| forEach | מבצעת פונקציה פעם אחת על כל האיברים במערך (לא מחזירה כלום) |  |
| Map | מפעילה פונקציה שרוצים ומפעילה אותה על כל איבר במערך. מחזירה את התוצאות במערך חדש (כלומר צריך להשתמש בערך המוחזר, כי לא משנה את המערך המקורי) |  |
| filter | שיוצרת עותק רדוד של חלק ממערך נתון, ומסננת אותו רק לאיברים שמקיימים את התנאי המיושמת על ידי הפונקציה שסופקה. [לשים לב אי אפשר להחזיר ערך אחר משם אלה מקבלים את המערך המקורי מקוצץ איברים] |  |
| reduce | מבצעת פונקציית callback על כל איבר של המערך, על מנת להעביר את ערך ההחזרה מהחישוב מהאלמנט הקודם לאיבר הבא. התוצאה הסופית היא ערך בודד. |  |
| find |  |  |

פונקציה flat, מורידה מערך מתוך מערך ככה שהכל יהיה פרוס יחד במערך אחד מרכזי, ואם יש כמה מערכים אחד בתוך שני בריבוי, אפשר לתת פרמטר של מספר, והוא פשוט יפרק עומק למשל של 5 מערכים למערך יחיד.

Js ו-DOM

בשלב זה נשים את ה-JS בסוף ה-IMPORT-ים שבקובץ HTML.

const header = ducument.getElementById("main-header");

נקפיד שה-ID יהיה יחודי (אסור שם שחוזר על עצמו) נעבוד על משתנה כי אחרת זה עבודה עם קוד לא בצורה מסודרת.

const pargraph = ducument.getElementByTagName("p); יחזיר לנו קולקשיין שנקרא HTML COLLECTION.

const intro = ducument.getElementByTaClassName("intro); יחזיר קולקשיין גם כן של כל האלמנטים מהמחלקה הזאת.

addEventListner("click", function(){...etc}) יוצר מאזין וזה אפשרי לכל אלמנט ב-HTMK.

paragraph[i].innerHTML = "changed"; מאפשר לשנות את הכיתוב של האלמנט הנ"ל.

נעבוד עם .srtAttribiute("class","nameclass")

ולא עם styleColor="blue" למשל

const elements = ducument.querySelectorAll(\*) ייקבל כנראה את כל האלמנטים בדף

יצירת אלמנט חדש בסוך הקוד HTML:

netElem = ducument.creatElemnt("p")

newElem.innerHtml = "this is name a elem"

const newDiv = getEelemntByID("newDiv");

newDiv.appendChild(newItem) שמנו למעשה בתוך ה-DIV הקיים את האלמנט החדש שיצרנו.

בעצם יצירת אלמנט מורכבת מ-:

1. יצירה (משתנה).

2.שמים משהו בתוכו בהשמה (inner\*)

3. לוקחים את האלנט מתוך הדף שבתוכו רוצים שזה יהיה

4. עושים appendChild בתוך האלמנט שרוצים שיהיה בתוכו.

const p3 = ducument.getElementById("p3");

p3.paentNode.removeChild(p3) מסיר את האלמנט מהדף.

איפה שיש החזרה של קולקשיין, זה לא אומר שעובדים עם pop וכו' אלה צריך לעבוד עם [].

אבל זאת הדרך שזה כן יהיה מערך:

const specials = document.getElementsByClassName('special');

const arr = [...specials];

arr.pop() //this works because it's now an array!

//To find the first li on the page:

document.querySelector('li');

// To find the first li with the class of special, nested inside of a ul, nested inside a section:

document.querySelector('section ul li.special');

// To find ALL li's on the page:

document.querySelectorAll('li');

const targetLI = document.querySelectorAll('li.todo')[2] //3rd li with class of 'todo'

element.prepend() - פונקציה שפועלת על אלמנט ומאפשר

להכניס מספר אלמנטים(כולל טקסט) ותמיד היא מכניסה אותה לראש הרשימה

של האלמנטים.

דרך נוספת להכניס אלמנט לפני אלמנט אחר זה ע"י:

parent.insertBefore(what to insert, where to insert)

בעצם רושמים אם האלמנט שרוצים להכניס ואחרי זה לפני איזה אלמנט רוצים להכניס.

//We call removeChild() on the parent element and pass in the element we want to remove:

removeMe.parentElement.removeChild(removeMe)

דרך נוספת להסרת אלמנט (לא עובד בכל הדפפנים).

הוספרת Attribute לאלמנט:

newLi.setAttribute('data-id', e.id) ;

שינוי סטייל לאובייקט בצורה חופשית:

ol[0].style.cssText = "list-style-type :none";

איך מקבלים את האלמנט הבא בתוך אלמנט האב?

this.nextElementSibling

כמובן שלא חייב את ה-this, הכוונה לאלמנט כלשהו (במקרה הזה this).

console.dir(nameelemnet) =תדפיס לקונסול את כל הפרטים של האלמנטים.

|  |  |
| --- | --- |
| **innerText** | **innerHTML** |
| We can not insert the HTML tags. | We can insert the HTML tags. |
| It ignores the spaces. | It considers the spaces. |
| It returns text without an inner element tag. |  |

-textContents הוא כל הטקסט הכלול באלמנט וכל הילדים שלו, שנועדו למטרות עיצוב בלבד.   
innerText - מחזירה את כל הטקסט הכלול באלמנט ואת כל רכיבי הצאצא שלו.   
innerHtml - מחזיר את כל הטקסט, כולל תגי html, הכלול באלמנט.

<div id="mylinks">

This is my <b>link collection</b>:

<ul>

<li><a href="www.borland.com">Bye bye <b>Borland</b> </a></li>

<li><a href="www.microfocus.com">Welcome to <b>Micro Focus</b></a></li>

</ul>

</div>

|  |  |
| --- | --- |
| browser.DomElement("//div[@id='mylinks']").GetProperty("textContents") | This is my link collection: |
| browser.DomElement("//div[@id='mylinks']").GetProperty("innerText") | This is my link collection:Bye bye Borland Welcome to Micro Focus |
| browser.DomElement("//div[@id='mylinks']").GetProperty("innerHtml") | This is my <b>link collection</b>:  <ul>  <li><a href="www.borland.com">Bye bye <b>Borland</b></a></li>  <li><a href="www.microfocus.com">Welcome to <b>Micro Focus</b></a></li> |

innerText innerHTML

We can not insert the HTML tags. We can insert the HTML tags.

It ignores the spaces. It considers the spaces.

It returns text without an inner element tag.

innerText - זה נטו טקסט לקריאה.

button.setAttribute("name", "helloButton");

input[type="password"] אפשר לעשות ב-CSS.

אם רושמים את הפונקציה הבאה ב-console עבור אלמנט מסוים, זה ייתן את כל המאפיינים שניתן לשים בסטייל עבורו:

Const c = **getcomputedstyle**(li);

c.color();//למשל

styles.color

מאפשר לקבל פרטים אודות האלמנט (למשל range)

getAttribute()

וזה מאפשר לקבוע ערך

setAttributf)'min','-500')

אם רוצים לגבי את הבן של אלמנט:

list.**children**[0]

list.**insertBefore**(newNode, list.**children**[0]);

מאפשר להכניס בתוך list את האלמנט newnode כאשר הוא יהיה בתוך הרשימה לפני list.children[0].

activeElem.insertAdjacentElement('beforebegin',tempDiv);

מאפשר להכניס אלמנט (tempDiv) במקום מסוים ביחס לאלמנט שעליו הוא נקרא (activeElem).

**כאשר האפשריות:**

<!-- beforebegin -->

<p>

<!-- afterbegin -->

foo

<!-- beforeend -->

</p>

<!-- afterend -->

הפונקציה:

Element. prepend(param1, param2, /\* … ,\*/ paramN)

השיטה Element.append() מכניסה קבוצה של אובייקטי Node או אובייקטי מחרוזת אחרי הילד האחרון של האלמנט. אובייקטי מחרוזת מוכנסים כצמתי טקסט מקבילים.

let div = document.createElement("div");

let p = document.createElement("p");

let span = document.createElement("span");

div.**append**(p);

div.**prepend**(span);//פועלת זהה רק שמה בהתחלה ולא בסוף

console.log(div.childNodes); // NodeList [ <span>, <p> ]

div.append("Some text", p)

console.log(div.childNodes) // NodeList [ #text "Some text", <p> ]

div.append("Some text");

div.prepend("Headline: ");

console.log(div.textContent); // "Headline: Some text"

קבלת אלמנט האבא

document.getElementById("myLI").parentElement

פונקציה שמחזירה את כל הילדים children:

for (const child of myElement.children) {

console.log(child.tagName);

}

**אירועים**

האירוע לא מתרחש למשל בכפתור, אלה מתרחק ב-document עצמו (באלמנט ה-root).

ובדרך עובר בשרשור למטה מאב לבן לנכד עד שמגיע לצאצא הרלוונטי באירוע.

ואז זה כאילו בפועל שהאירוע התרחש בכל האלמנטים ולא רק איפה שהגדרנו אותו.

ברירת מחדל של אירוע של bubling

ואם רוצים שיהיה הפוך והאירוע יתרחש מהאב ואז ישורשר לבן רושמים ככה:

('click', Function(event){}, true)

וזה משנה ל- capturing כלומר מאבא קודם מתרחש האירוע ובסוף לבן.

אבל לא משתמשים בשיטה הזאת.

כלומר אם יש לנו אירוע מסוים שמטופל בבן, וגם האבא יש לו פונקצית טיפול.

אז אפשר שגם הבן בפונקציה שלו יטפל באירוע וגם האבא כי שניהם מקבלים את אותו אירוע.

Function(event){}

event.target

כנראה מתייחס לכולם

event.currentTarget

מתייחס רק לאירוע הנוכחי (ה-this).

event.stopPorpagation()

הפעולה הזאת תפסיק את האירוע לכיוון האלמנטים האחרים בשרשור (נניח רוצים רק בבן, זה יהיה רק בבן ולא יבעבע לאבא).

לא משתמשים ב-stop כי מתערב בדפדפן וזה לא תקין לעבוד ככה.

Event.preventDefault()

שיטת preventDefault() של ממשק האירוע אומרת לסוכן המשתמש שאם האירוע לא יטופל במפורש, אין לנקוט בפעולת ברירת המחדל שלו כפי שהיא הייתה בדרך כלל.

נשים לב שבאירועים נשתמש ב-function ולא ב-arrow function בגלל הthis

בחלק של HTML אם רוצים לעשות ישירות מאזין עם קוד פשוט, אפשר לרשום:

Onclick="console.log()"

אפשר גם לרשום כך ממש בתוך הקוד של JS:

btn.onclick = function(){...etc}

window.screen

אם נרשום בקונסול, נקבל מידע מלא לגבי התצוגה של המסך.

הוספת מחלקה לאלמנט:

elem.classList.add("nameclass");

אם רוצים לפרק ממערך נתון לכמה משתנים אפשר לעשות דבר כזה:

const { name, type, value, checked } = target;

אם עושים מאזין לחלון, זה מאזין ברמת החלון לכל לחיצה שנלחצת:

window.addEventListener

**this:**

הסוגים של this:

סוג אחד עם 4 תתי סוגים ונקרא invocation:

1. function

אז אם ה-this נמצא שם, סימן שזה this==global

בעצם זה אומר שהוא מקבל את הסקופ של מחוץ לפונקציה ה-this הזה בתוך הפונקציה.

כלומר זה נמצא באובייקט הגלובלי של js

וסביבה של הדפדפן זה ב-window

-מאותה סיבה גם ה-this של מחוץ לפונקיה שווה ל- this===global.

2. אם עושים בתוך הפונקציה use strict

אז ה-this יהיה undifned.

גם שיש פונקציה בתוך פונקציה ה-this בפונקציה הפנימית יהיה במקרה הזה

undefined.

3.

באירוע כמו keypreess אז this.value זה הערך שהוכנס עכשיו

new מקשר בין הקונסט בין המחלקה למשתנה ואם לא נרשום new למחלקה זה יעבוד אבל

אין לנו קישור של ה-this

בלי New המופע מצביע לגלובל ואם עובדים עם new ה-this לא מצביע לגלובל.

זה במחלקות.

בעצם בלי new מקבלים הצבעה לגלובל, זה רלוונטי לראיונות עבודה.

קונטקס - מה יהיה הערך בסביבה מסוימת.

הארגומנטים של call זה כמו שהם ואילו ב-applay מקבל פרמטרים שבתוך מערך אחד.

הם עושים אותה פעולה call ו- apply בערך.

function(num){ return this\*num}

const double = mul.bind(2);

double(2)

בעצם הדאבל הפך להיות סוג של מצביע לפונקציה, ככה שעכשיו קריאה עם double

קורא ל-mul אך הפעם עם 3.

3 הפונקציות הבאות עושות אותו הדבר בדרך מעט שונה:

CALL ו- APLLAY מקשרות קונטקס ועושה INVOKE.

ו-bind מחזירה פונקציה חדשה ו-הthis שלה מקושר לאויביקט המקורי שעליו עשינו bind

להוסיף את הדרכים שיש כדי לעשות השמות לאובייקט כמו מתרגיל 12.2 אתמול.

היום פחות משתמשים בזה בתעשייה, אלה משתמשים בפונקצית arrow.

Bind עושה קישור בין המשתנה לבין ה-this באופן קבוע (לזה הראשון שקראנו לו עם bind ואחרי זה כל קריאה

עם המופע הזה הקישור יהיה על הרפמטר המקורי שמקשור ל-this כמו ב-mul ה-this תמיד יהיה 2 שעליו עשינו במקור

את ה-bind.

למשל –

    const s = printSuperPower.bind(storm)

    s();

בעצם יצרנו קישור עבור הפונקציה כך שהפרמטר שהועבר דרך bind זה יהיה מה ששוה this בפועל.

function test(arg1, arg2){  
  console.log(this.num, arg1, arg2); // 100, 10, 20  
}  
  
test.call({num: 100}, 10, 20);

function test(...arguments){  
  console.log(this.num, arguments);//100, [1,2,3]  
}  
  
test.apply({num: 100}, [1,2,3]);

function test(arg){  
 console.log(this.number, arg);  
}  
  
let bindedFn = test.bind({number: 99}, "argument");  
  
bindedFn(); // 99, "argument"

ההבדל בין כולם- call ו-applay מקשרות משתנים/מערך בהתאמה וגם קוראת מיד לפונקציה.

Bind מקשרת משתנים אך לא קוראת לפונקציה, וקוראים לה בפעם אחרת מתי שנרצה.

אפשר לראות בקוד הבא, לפני שהוסיף את ה-that, ה-this היה בתוך render ואילו שהיתה קריאה ל-asyncGet ע"י המצביע, זה מדבר כבר על ה-this הנוכחי שלו ולא על ה-this של render.

הדרך שפה זה ע"י תיקון של שמירת למשתנה מקומי, ואז זה נדבק אליו.

אבל זה לא הפתרון הרצוי, כי רוצים את this. ולכן בפתרון אחרי יש חיבור של הפונקציה שמתקבל כפרמטר אז יש הדבקה של this ע"י bind ישירות ככה שתהיה קריאה עם המצביע, ה-this למעשה קושר אליה.

שימושים של call, bind, applay.

יש לנו למשל מערך ורוצים למצוא מינימום, במצב רגיל min של math לא מקבלת מערך

אז עושים math.min.apply(null, arr)

let mammal = function(legs){

    this.legs = legs;

}

let cat = function(legs, dim){

    console.log(this)

    mammal.call( this , legs);//רוצים לקשר את דִיס ככה ממל הדיס שלו יהיה של קאט

//ולכן יתווסף שדה של לֶג לקאט

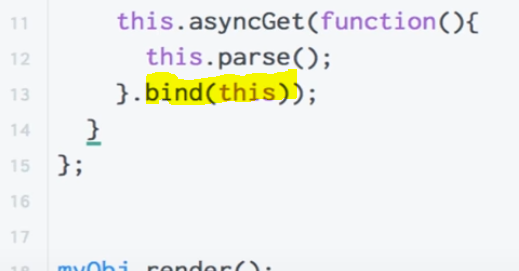
    this.dim=dim;

}

let lion = new cat(4, false)

console.log(lion)





ב-arrow func ה-this שם יהיה שווה למופע (בעצם יורש מה שנמצא מעליו).

אבל ב-function רגיל ה-this הוא של הגלובל.

בעצם שה-this הוא גלובל אז, ה-this רואה למעשה את המשתנים מחוץ לאותו איזור ולא מוגבל למשל רק למשתני המחלקה.

לסיכום ה-THIS בכל מקרה:

1. בפונקציה - הוא גלובלי.

2. במתודה - הוא שווה לאובייקט.

3.בבנאי - הוא שווה למופע.

4. וב-call,applay, bind - הוא שווה לקונטקסט של האובייקט (שנמסור לו).

5. arrow func- הוא שווה ל- outer קונטקס.

ניתן לראות שה-this מצביע לתחילת הבלוק של הפונקציה (אובייקט) כאן:

const person = {

first: 'Cherilyn',

last: 'Sarkisian',

nickName: 'Cher',

fullName() {

//In a method, this refers to the object the method "lives" in

const {

first,

last,

nickName

} = this;

החלק הזה זה המשך של אותו אובייקט, אם נעשה קריאה ע"י

Person.printBio() נקבל שה-this מצביע לחלק של האובייקט (למשתנים שלו)

לעומת זאת אם נעשה person.laugh() נקבל שה-this מצביע ל-{} (כי שם זה arrow function וזה שונה בפונקציה כזאת לעומת רגילה).

הthis של laugh הוא לא גלובלי, הוא undefined כי בשביל להפעיל את המתודה ולקבל את הערך של this צריך להוציא אותה מחוץ לפונקציה ואז הערך של this הולך לאיבוד

הערה – ולכן לא נשים arrow function בתוך אובייקט ברוב המקרים.

printBio() {

console.log(this);

const fullName = this.fullName();

console.log(`${fullName} is a person!`)

},

laugh: () => {

//Arrow functions don't get their 'own' this.

console.log(this);

console.log(`${this.nickName} says HAHAHAHAH`)

}

}

בדוגמא הבאה, נראה כאשר יש לנו בתוך הפונקציה של start קריאה ל- setInterval

אז אם הקריאה היא ע"י arrow function כמו כאן למטה, אז ה-this מתיחס לאובייקט ( הוא מתייחס לאובייקט כי הוא מקבל את הקונטקסט בירושה).

ואם היינו שמים במקום arrow פונקציה אנונימית בצורה של function {} ה-this היה גלובלי (אפילו שזה בתוך start, אבל ה- setInterval לעומת זאת יושב במקום הגלובל ולכן זה לא היה עובד כמצופה.

const annoyer = {

phrases: ["literally", "cray cray", "I can't even", "Totes!", "YOLO", "Can't Stop, Won't Stop"],

pickPhrase() {

const {

phrases

} = this;

const idx = Math.floor(Math.random() \* phrases.length);

return phrases[idx]

},

start() {

//Use an arrow function to avoid getting a different 'this':

this.timerId = setInterval(() => {

console.log(this.pickPhrase())

}, 3000)

ואם נשים בתוך מאזין:

submitBtn.addEventListener("click",() =>

אז ה-this יהיה גלובלי שזה aroow בתוך אירוע.

ואילו אם נרשום ככהף

submitBtn.addEventListener("click", function (event) {

זה יהיה ה-this של המופע הנוכחי (שזה הכפתור submit ).

מחלקות.

דרכים ליצירת אובייקטים:

object.creat - פחות בשימוש ליצירת אובייקטים

factory - ע"י בנאי שמתחיל באות גדולה.

const Car = function(name, manu){

Vehicle.call(this, name);

this.\_menu=menu;

this.hetMenu = function(){return \_menu}};

};

const honda = new Car("honda civic", "honda");

3. class- זה השימש העיקרי ליצירת אובייקטים.

class Person{

constructor (fullName, birthYear)

} this.fullName = fullName;

This. birthyear = birthyear;

}

calcAge(){

console.log(2023 – this.brithYear);

get age(){ return 2023 – this.brithYear;}

set fullName(name){ this..fullName = name;}

static getAll(){…}

}

Const rami = new Person ("rami levi" 1968);

rami.fullName = "rami choen";//הוא עושה את זה דרך הסטר למרות שאין סוגריים בקריאה

Person.getAll();// קריאה למתודה סטטית ישירות דרך שם המחלקה

מתודה סטטית בתוך מחלקה יושבת על הפרוטוטייפ ולא כל ה-class

Class Student extends Person {

Constructor Person(fullName, brithYear, course){

Super(fullName, brithYear);//קריאה לאב

This.course.course;

}

calcAge(){ …} //מימוש משלו למתודה וזה פולימורפיזם

}

Class Account {

#pin; //private

#movements; //private

Owner;

Constructor (owner, pin){

this.owner = owner;

}

set pin(pin){

this.#pin = pin;

}

Deposit(val){

this.#movements.push(val);

return this;//מחזיר את המופע הנוכחי ומאפשר לממש את השורה הצהובה למטה

}

#approveLoan(val) {…)//מתודה פרטית

}//class

Const acc = new Account("mazal","ils");

acc.#pin = 1234;//זה אסור כחי לא נראה משתנה פרטי מבחוץ

acc.pin =1234;//אפשרי וזאת קריאה לסטר

acc.deposit(300). deposit(500).withdraw(35).requestLoan();

בגלל שמימשנו שחזירים את המופע הנוכחי (this) זה מאפשר לשרשר קריאה, כי כל קריאה מחזירה את המופע הנוכחי.

לימוד מהמשחק snake:

אם עושים בקובץ input.js את הדבר הבא בפונקציה:

export function functionName() { /\* … \*/ }

מאפשר לייצא את הפונקציה בקוד ולאפשר לקובץ אחר לעשות לה ייבוא.

import { functionName, func2 } from "./input.js"

אפשר גם לרשום ככה כדי לקצר שמות:

import { update as updateFood, draw as drawFood } from './food.js'

Window.requestAnimationFrame()

שיטת window.requestAnimationFrame() אומרת לדפדפן לבצע הנפשה ומבקשת שהדפדפן לקרוא לפונקציה שצוינה כדי לעדכן אנימציה לפני הצביעה מחדש הבאה. השיטה לוקחת התקשרות חוזרת כארגומנט שיש להפעיל לפני הצביעה מחדש.

היא נקראת בממוצע 60 פעמים בשנייה אחת.

const element = document.getElementById('some-element-you-want-to-animate');

let start, previousTimeStamp;

let done = false

function step(timestamp) {//זאת חותמת זמן שהגאווה ממלא כל קריאה

if (start === undefined) {

start = timestamp;

}

const elapsed = timestamp - start;

if (previousTimeStamp !== timestamp) {

// Math.min() is used here to make sure the element stops at exactly 200px

const count = Math.min(0.1 \* elapsed, 200);

element.style.transform = `translateX(${count}px)`;//הזזת אלמנט בציר ה-איקס

if (count === 200) done = true;

}

if (elapsed < 2000) { // Stop the animation after 2 seconds

previousTimeStamp = timestamp;

if (!done) {

window.requestAnimationFrame(step);

}

}

}

window.requestAnimationFrame(step);

איך בנוי בבסיס משחק זמן אמת?

תהיה קריאה כזאת:

window.requestAnimationFrame(main)

כאשר זאת הרוטינה העיקרית:

function main(currentTime) {

  if (gameOver) {

    if (confirm('You lost. Press ok to restart.')) {

      window.location = '/'

    }

    return

  }

  window.requestAnimationFrame(main)//סוג של לולאת עידכון נתונים במסך

  const secondsSinceLastRender = (currentTime - lastRenderTime) / 1000

  if (secondsSinceLastRender < 1 / SNAKE\_SPEED) return

  lastRenderTime = currentTime

  update()//ואז קודם עידכון נתונים

  draw()//ואחרי זה עידכון תצוגה ככה זה בלופ

}

למשל סנייק, יש לנו UPDATE שאחראי לעדכן את מבנה הנתונים

פונקצית DRAW שאחראי לעדכן את החלק של התצוגה – לצייר במסך.

הפקודה

window.location = '/'

גורם להפניה של הדף ל-URL שנבחר.

אם שולחים לפונקציה ארגומנטים ובמקור הפונקציה לא אמורה לקבל פרמטרים, יש אפשר לגשת לארגומנטים בצורה הבאה:

function test(){

    console.log(arguments)// { '0': 1, '1': 2 }

    console.log([].slice.call(arguments));//[1, 2]

}

test(1,2)

const bodyBoundary = document.body.clientWidth;//רוחב המסך

const elRight = element.getBoundingClientRect().right;//מיקום מימין סופי

const currLeft = element.getBoundingClientRect().left;

setTimeout(() => {},timeToExe}//

**REST API**

**API** (Application Program Interface) הוא דרך מוסכמת לשלוח ולקבל מידע בין מחשבים. לדוגמה, אם אתה מעוניין להציג באתר שלך מפות גוגל אבל (לצערך) המפות לא נמצאות על השרת שלך אלא על השרתים של גוגל. אז הדרך לבקש מגוגל לשלוח לך את המפה המבוקשת היא באמצעות **API** שמספק נותן השירות (גוגל) שאומר לאילו כתובות אינטרנט צריך להפנות את הבקשות כדי לקבל את המידע. הבקשה היא ה-**request** ששולח האתר שלנו, והתגובה המוחזרת מהשרתים של נותן השירות היא ה-**response**.

פעולה אסנכירות תחשב רק כאשר:

הקוד ממשיך לפעול ברקע.

JSON זה אובייקט של JS פעם היה XML אבל זה מה שעובדים ב-API-ים וכו'.

זה למעשה STRING.

JSON FORMATER תוסף כנראה לכרום מראה בצורה יפה את המבנה במקום בשורה אחת.

מבקשים נתונים המשרת בצורת HTTP מקבלים JSON

וזה נעשה באמצעות REST

שזה סט של פונקציות שמשתמשים מתכנתים בהם כדי לקבל נתונים וכו'.

לדפדפן יש API שנקרא FEATCH ומאפשר לנו לדבר עם שרתים.

fetch זה פונקציה אסינכרונית.

היא לוקחת את הנתונים ומביאה אותם מהשרת (API).

טכנית היא מחזירה PROMISE (זה למעשה אובייקט של תוצאה של פעולה אסניכרונית, או סוג של קונטיינר לערך עתידי).

יש לזכור ש-fetch זה מחכה לערך לדבר שקורה עכשיו, ואילו אירוע מאזין, כרגע לא עושה כלום עד שלמשל לא תהיה לחיצה.

**PROMISE**

Promise הוא אובייקט של JavaScript שבתוכו אנחנו שמים קוד, שמחזיר אחת משתי תוצאות, הצלחה או כישלון.

האובייקט Promise **מקבל פונקציה בתור פרמטר**, והפונקציה הזו מקבלת שני פרמטרים: resolve ו-reject.

new Promise((resolve, reject) => {});

resolve - שתקרא במקרה שהקוד הצליח.

reject - שתקרא במידה והקוד נכשל.

יש צורך לקלוט את התגובה באמצעות המתודות **then** ו-**catch**. המתודה then קולטת את ה - resolve, והמתודה catch קולטת את ה-reject.

דוגמא:

let flipCoin = new Promise((resolve, reject) => {

// The code

// Resolve in the case of success

if(…){

resolve("Yeah, it's head!");

}

// Otherwise reject

reject("No luck this time!");

});

flipCoin

.then(fromResolve => console.log(fromResolve))

.catch(fromReject => console.log(fromReject));

-מה שצבוע בצבע תכלת, זה פשוט סינטקס של קריאה לפונקציה, כלומר אפשר לרשום איזה שם שרוצים (כי זה שם הפרמטר שלה) וזה לא קשור לשם הפונקציה בפועל. פשוט הכתיב הראשון תמיד מתייחס לקריאה reolve והשנייה לקריאה מ-reject.

לסיכום לפי ההבנה שלי – promise זאת קריאה לפונקציה רגילה למעשה, עכשיו אם הפונקציה הצליחה, ה-promise למעשה ברמת הקריאה (בלי בדיקות של if וכו') מיד תתפוס את התשובה ע"י resolve ותרשום מה שצריך, ואם הפעולה של גוף הפונקציה נכשלה אז הפונקציה תחזיר תשובה ע"י reject את התשובה שם.

בחיי היום-יום אנחנו משתמשים ב- promise עבור קוד א-סינכרוני. לדוגמה, קריאה ב-ajax ל-API חיצוני או טעינת תמונה ל-DOM

דוגמא לשימוש:

במקור שרוצים לעשות שירשור של פונקציות שכל אחת תלויה בתוצאה של זאת מעליה, היינו אמורים לעשות את זה ככה (אך לא נעשה, בגלל הצורה הלא קריאה):

talkWithTheCustomer((input) => {

workOnTheProject((result) => {

deliverOnTime((finalResult) => {

console.log('Finito!');

}, failureCallback);

}, failureCallback);

}, failureCallback);

וזה הפתרון ע"י פרומיס:

talkWithTheCustomer(input)

.then(result => workOnTheProject(result))

.then(result => deliverOnTime(result))

.then(finalResult => console.log('Finito!'))

.catch(fromReject => console.log('Error'));

let talkWithTheCustomer = (input) => {

return new Promise((resolve, reject) => {

window.setTimeout(() => {//כדי לדמות אסינכרוניות כמו המתנה מאֶפִאַיי של שרת

if(input === 'agree') {

resolve("OK");

}

reject("Sorry, non starter");

}, Math.random()\*1000);

});

}

באופן דומה הפונקציה workOnTheProject כתובה.

ויש דגש חשוב, שחייב שכל פונקציה בעצמה תחזיר פרומיס (כי זה כמו שרשור פרומיס, ובעצם אנו מפעילים את הפרומיס הבאה על הפרומיס הקודם, וזאת הסיבה שאפשר לרשום בקריאה then ו-catch.

כל עוד ה-promises מתקיימים, הקוד ממשיך ועובר מפונקציה לפונקציה לפי הסדר של השרשור עד לסיום, אבל במקרה שאחת ההבטחות כושלת, הקוד מפסיק, וקופץ למתודה catch שתופסת פיקוד. כדי לראות כיצד זה פועל נסו לשנות את הערך של המשתנה input , ואז תראו שה-catch הוא זה שמגיב עם ההודעה שמספק ה- reject של הפונקציה הראשונה, והפונקציה השנייה בכלל לא רצה.

אפשר למשל להחזיר במקרה של הצלחה דרך ה-resolve אובייקט:

resolve({ status: 200, data });

כאשר הגישה ל-data למשל תהיה ככה:

    .then((res) => {

        console.log('Status Code', res.status);

כלומר ברמה ה-then (בקריאה המקורית לפונקציה) של יש את התוצאה ב-res שזה אובייקט שמכיל את כל הנתונים שהוחזר, למשל גישה לערך ה-status תהיה ע"י res.status.

הערה- יש לשים לב שברמה של ה-then ו-catch יכול להתבצע קוד ממשי בבלוק הזה ממש כמו פונקציה לכל דבר.

יש אפשרות גם לחכות לשני אירועים ע"י all.

const api = 'https://phpenthusiast.com/dummy-api/';

let dice1 = fetch(api);//מחזירה פרומיס

let dice2 = fetch(api);//מחזירה פרומיס

Promise.all([dice1,dice2]);//המתנה למספר אירועים שיסתיימו

בהתחלה ה-PROMISE במצב של PENDING לפני שיש את הערך שנצרך.

ברגע שהמשימה הסתיימה להיות במצב SETTLED ואז זה יכול להיות FULFILLED שזה הצליח או REJECTED וזה לא הצליח.

גם סינטקס של פונקציה אנונימית וגם טיימר

btn.getBoundingClientRect();//תיתן לנו מקום של האלמנט במסך

const p = new Promise((resolve, reject) => {

    …

    if (..) {

        resolve();

    }

    else {

        reject();

    }

});

p.then(() => {

…

});

p.catch(() => {

…

});

**מימוש promise ע"י await ו-async.**

אז מה זה Async/Await?  
חשוב לציין – Async/Await **לא מחליף את ה-promises,** בסך הכל מדובר בצורת כתיבה נוחה יותר למימוש promises מאשר ה-then המשורשר (syntactic sugar).  
מדובר על פונקציה שבתחילה **אנו קובעים שהיא פונקציה מסוג async** על ידי כתיבה המילה "async" לפני הכרזתה ובתוך הפונקציה, **לפני מימוש של כל Promise אנו "מחכים" ל-Promise באמצעות המילה "await". בתוך הפונקציה ממש.**

const dice = () => {

return fetch(api).then(res => res.json());

const diceThrower = **async** () => {

let dice1 = **await** dice();

}

לא יתבצע הקוד שאחרי await עד שהשורה לא תתבצע וזה רלוונטי רק על הפונקציה הזאת של ה-async.

זה חייב לבוא בחייב אי אפשר לרשום await בלי async.

הערה- יש לשים לב, שאם יש כמה פקודות בגוף הפונקציה שרוצים לחכות, לכל אחת צריך לרשום await.

לא הבנתי למה, אבל פה מראים שני מקרים שבאחד מקבלים await/async שפועל פקודה פקודה ואחד נוסף באופן מקבילי:

// SEQUENTIAL REQUESTS!

async function get3PokemonSequential() {

    const poke1 = await axios.get('https://pokeapi.co/api/v2/pokemon/1');

    const poke2 = await axios.get('https://pokeapi.co/api/v2/pokemon/2');

    const poke3 = await axios.get('https://pokeapi.co/api/v2/pokemon/3');

    console.log(poke1.data);

    console.log(poke2.data);

    console.log(poke3.data);

}

// PARALLEL REQUESTS!

async function get3Pokemon() {

    const prom1 = axios.get('https://pokeapi.co/api/v2/pokemon/1');

    const prom2 = axios.get('https://pokeapi.co/api/v2/pokemon/2');

    const prom3 = axios.get('https://pokeapi.co/api/v2/pokemon/3');

    const poke1 = await prom1;

    const poke2 = await prom2;

    const poke3 = await prom3;

    console.log(poke1.data);

    console.log(poke2.data);

    console.log(poke3.data);

}

ה-await בעצמו תמיד מחזיר פרומיס, גם אם הפעולה שהוא מבצע לא מחזירה פרומיס, הוא עצמו מוגדר להחזיר פרומיס.

הערה – אם יש שורה אחרי ביצוע הקריאה לפונקציה שהיא async (לא בתוך גוף הפונקציה, אלה ב-Main כביכול) זה ימשיך להתבצע רגיל עד שהקריאה של ה-await תחזיר תשובה.

כלומר זה קוד אסינכרוני ואילו שאר הקוד הוא מתבצע כרגיל בזמן הזה.

לעומת זאת הקוד שבא מיד אחרי ה-await (כלומר כל השורות שמיד אחרי בגוף הפונקציה שלה) לא יתבצעו עד שה-await לא תחזור.

עכשיו גם ב-await/async צריך להשתמש ב-catch/then ואפשר גם בדרך אחרת, להשתמש ב-try/catch בדרך הבאה:

async function getPlanets() {

    try {

        const res = await axios.get('https://swapi.co/api/planeklsajdalksts/');

        console.log(res.data);

    } catch (e) {

        console.log('IN CATCH!', e);

    }

}

getPlanets();// קריאה רגילה בלי then

אם נרצה להוסיף async לפונקציה של arrow נרשום את זה ככה:

const retrieveRecipe = async () => {

כמו כן, יש לשים לב שרוצים להוסיף async/await ויש פונקציות שמחזירות את הפרומיס, לא נוסיף לקריאות esolve/rejectאת הפקודה await אלה נעטוף קריאה של פונקציה חדשה שהיא תהיה זאת שקוראת לפונקצית שמחזירה פרומיס מ-resolve.

אם הפונקציה זורקת throw, אז זה יתבטא ב-reject.

בעצם מה שצריך לעשות זה לקרוא לפונקציה בצורה רגילה, ואז אפשר בתוך הפונקציה לעשות try על await ו-catch או בקריאה לפונקציה לעשות פשוט throw (על מה שתחזיר הפונקציה אם תהיה שגיאה).

למעשה await חוסך את הקריאות ל-then אבל עדיין צריך לעשות catch.

קוד מקבילי של fetch:

// PARALLEL REQUESTS!

async function get3Pokemon() {

  const prom1 = axios.get('https://pokeapi.co/api/v2/pokemon/1');

  const prom2 = axios.get('https://pokeapi.co/api/v2/pokemon/2');

  const prom3 = axios.get('https://pokeapi.co/api/v2/pokemon/3');

  const results = await Promise.all([ prom1, prom2, prom3 ]);

  printPokemon(results);

}

function printPokemon(results) {

  for (let pokemon of results) {

    console.log(pokemon.data.name);

  }

}

get3Pokemon();

// SEQUENTIAL REQUESTS!

async function get3PokemonSequential() {

    const poke1 = await axios.get('https://pokeapi.co/api/v2/pokemon/1');//עד שלא מסיים לא עובר לפקודה הבאה

    const poke2 = await axios.get('https://pokeapi.co/api/v2/pokemon/2');

    const poke3 = await axios.get('https://pokeapi.co/api/v2/pokemon/3');

    console.log(poke1.data);

    console.log(poke2.data);

    console.log(poke3.data);

}

// PARALLEL REQUESTS!

async function get3Pokemon() {

    const prom1 = axios.get('https://pokeapi.co/api/v2/pokemon/1');//כל קריאה מחזירה פדינג

    const prom2 = axios.get('https://pokeapi.co/api/v2/pokemon/2');

    const prom3 = axios.get('https://pokeapi.co/api/v2/pokemon/3');

עכשיו בעצם הבקשות כולם נשלחו במקביל, ונשאר כדי לבצע את ההשמה של התוצאה, ולכן נרשום await שזה כבר החלק הסופי של ההשמה יעשה באופן טורי ולא מקבילי (חייב לשים במשתנה כי אם לא, בגלל שלא עבדנו עם await ב-fetch ולכן יש לנו למעשה pending ואם נגיע לשורות של ה-logלמעשה בשורה ההיא היתה השמה של פרומיס שהוא פדינג, ועוד לא התבצע השמה של התוצאה, ולכן חייב לעשות השמה עם יחד עם await.

    const poke1 = await prom1;//בכל

    const poke2 = await prom2;

    const poke3 = await prom3;

    console.log(poke1.data);

    console.log(poke2.data);

    console.log(poke3.data);

}

get3Pokemon();

הערה –

Axios היא ספריית HTTP מבוססת הבטחות המאפשרת למפתחים להגיש בקשות לשרת שלהם או לשרת של צד שלישי כדי להביא נתונים. הוא מציע דרכים שונות להגשת בקשות כגון GET , POST .

הפונקציה fetch מחזירה בעצמה promise.

fetch('https://phpenthusiast.com/dummy-api/')

.then((res) => console.log(res))

.catch((error) => console.log(error));

ביצוע פעולת Post של הוספת רשימה ל-API

fetch('https://jsonplaceholder.typicode.com/posts', {

method: 'POST',

body: JSON.stringify({

title: 'foo',

body: 'bar',

userId: 1,

}),

headers: {

'Content-type': 'application/json; charset=UTF-8',

},

})

.then((response) => response.json())

.then((json) => console.log(json));

זה יחזיר את התשובה הבאה:

{

id: 101,

title: 'foo',

body: 'bar',

userId: 1

}

באופן דומה אפשר לעשות עידכון לאובייקט קיים ב-API ע"י כיתוב זהה רק ע"י PUT

ובנוסף לשים את ה-ID המתאים בתוך האובייקט שידע היכן לעדכן וזה מחייב כי חייב לרשום את כל השדות של האובייקט ובכל מקרה צריך לרשום כתובת מלאה שבסוף זה ה-ID ככה:

https://jsonplaceholder.typicode.com/posts/1

אפשריות של תיקון די דומה ל-put רק שב-put חייב לציין את כל השדו ואילו ב-patch מעדכן רק את השדה שרוצים ולא צריך לרשום את כולם.

fetch('https://jsonplaceholder.typicode.com/posts/1', {

method: 'PATCH',

body: JSON.stringify({

title: 'foo',

}),

headers: {

'Content-type': 'application/json; charset=UTF-8',

},

})

.then((response) => response.json())

.then((json) => console.log(json));

מחיקת אובייקט

fetch('https://jsonplaceholder.typicode.com/posts/1', {

method: 'DELETE',

});

זה הבדל בין XML ל-אובייקטים ב-JS שזה למעשה JSON (אוביקט JS נקרא גם JSON)



swapi.co/api

**xmlhttprequest**

זאת הדרך המקורית שמעבירים בקשות בגאווה

אבל זה לא תומך בפרומיס ולכן יש הרבה קריאות כלבק.



זה המבנה של בקשת XML פה יש שרשור קריאות (נשים לב שהבקשה השנייה היא בתוך המאזין של הראשונה כבקשת המשך):

const firstReq = new XMLHttpRequest();

firstReq.addEventListener('load', function() {//יש להוסיף מאזין

    console.log('FIRST REQUEST WORKED!!!');

    const data = JSON.parse(this.responseText);//פרמוט הפלט לגסון

    const filmURL = data.results[0].films[0];

    const filmReq = new XMLHttpRequest();

    filmReq.addEventListener('load', function() {

        console.log('SECOND REQUEST WORKED!!!');

        const filmData = JSON.parse(this.responseText);

        console.log(filmData.title);

    });

    filmReq.addEventListener('error', function(e) {

        console.log('ERROR!!', e);

    });

    filmReq.open('GET', filmURL);

    filmReq.send();

});

firstReq.addEventListener('error', (e) => {

    console.log('ERROR!!!!!!');

});

firstReq.open('GET', 'https://swapi.co/api/planets/');

firstReq.send();

console.log('Request Sent!');

then(res => if(!res.ok) throw new Error(res.status);

...catch(err => console.error(..)

**קבצי js:**

אם script נמצא בתוך ה-head

אז נחכה ונראה דף לבן עד שה-script לא יטען.

defer אומר לו לא לחכות ובינתיים להטעין את ה-HTML ובמקביל עובד על ה-JS

ושהוא מוכן הוא עושה את זה בסיום

אם שמים בלי defer בסוף ה-body זה יעבוד, אבל לאינטרנט שלא מהיר, זה יגרום לבעיות.

עכשיו יש defer ו async

כאשר רק defer עובד בדפדפן ישן.

async טוען ראשון קודם את הקבצים הקטנים שיש אינטרנ איטי.

defer מטעין את כל הקבצים במקביל.

שורה תחתונה defer עדיף על async