



Java Script

מקורות מידע חיצוניים:

- [גוגל AI - ג'ימיני](#)
- [קורס תכנות יוטיוב - עונה 5 עופר שלי](#)
- [קורס JS עם סהר טוויטו - Coding With Saar](#)
- [מכללת האקריו - קורס בניית אתרים](#)

מבוא ותחביר

Java script היא שפה דינמית הפופולרית בעולם המאפשרת בעיקר לפיתוח אתרי אינטרנט שפותחה ע"י גוגל ו- react שפותחה ע"י פייסבוק גם הן משתמשות בJS היא מתאימה לכל סוגי הדפדפנים וניתן לפתח בעזרתה אתרים WEB ואפליקציות עבור IOS ואנדרואיד. השפה מאפשרת להוסיף התנהגויות דינמיות לדפי אינטרנט סטטיים מה שהופך את חווית המשתמש באינטרנט למעניינת ומורכבת יותר.

כללים חשובים לשפה:

1. **הפרדה באמצעות נקודה פסיק:** כל שורת פקודה מסתיימת בנקודה פסיק;
2. **הקלדה דינמית:** סוג המשתנה נקבע בזמן ריצה ולא בזמן ההצהרה.
3. **רגישות לאותיות קטנות וגדולות:** JS רגישה לאותיות קטנות וגדולות כלומר name ו- Name אלו משתנים שונים.
4. **הערות:** ניתן להוסיף הערות בקוד באמצעות // עבור שורה אחת או /* */ עבור בלוק.
5. **מזהים:** מזהים (שמות משתנים, פונקציות וכו') חייבים להתחיל באות או תו תחתון (_) ולא יכולים להכיל תווים מיוחדים.

כשיש לנו משימה או מסמך איפיון לפיתוח אנחנו ניגש לפתח שלב אחרי שלב נחלק את הפעולות שיש לבצע עד לקבלת התוצאה הסופית.

שלב 1 – JS BASIC

Script

את קוד הJS ניתן להגדיר בתוך תגית סקריפט בHTML.

```
<script>console.log('hello world');</script>
```

בנוסף ניתן לקשר קובץ SCRIPT עם סיומת JS לתוך תגית SCRIPT אותה נכניס בתגית head של המסמך, לדוגמא:

```
<script src="./app.js"></script>
```

defer



כאשר אנו מקשרים קובץ JS ל HTML הדפדפן יטען את הקובץ JS בצורה דיפולטיבית מיד עם טעינת העמוד ובכדי למנוע זאת ולגרום לדפדפן לטעון תחילת את קובץ HTML ולאחר מכן את קובץ הסקריפט נשתמש במילה defer בתגית HTML בצורה הבאה:

```
<script src="./app.js" defer></script>
```

מתודות JavaScript

innerHTML – מבצעת כתיבת מהתוכנית ישירות לתוך HTML ניתן להוסיף ערכים תגיות ועוד.

משתנים

משתנים הם כמו קופסא שאפשר לשים בתוכה כל מיני דברים כגון: מספרים, מחרוזות בוליאנים, אובייקטים מערכים. ב JavaScript אנחנו נעזרים במשתנים כדי לאחסון מידע שישמש בתוכנית שלנו ניתן להגדיר אותם באמצעות var, let, const כעת נסביר על ההבדלים:

Var משתנה ניתן לשינוי בעל היקף רחב זמין גם מחוץ לקוד בלוק ספציפי פחות מומלץ לשימוש עצם התנהגותה הבלתי צפויה.

Let משתנה ניתן לשינוי, בעל היקף מקומי במידה והוגדר בתוך קוד ספציפי כמו פונקציה הוא אינו זמין מחוצה לה.

Const משתנה קבוע שלא ניתן לשינוי, גם הוא בעל היקף מקומי.

ניתן להגדיר כמה משתנים עם תו הפרדה ",", "בלי לחזור על מילת ההצהרה על משתנה חדש לדוגמא:

```
let x = 0, y=1;
```

בדוגמא הזו הצהרנו על 2 משתנים שונים הראשון X שיהיה שווה ל-0 והשני Y שיהיה שווה ל-1.

Let VS Var – על מנת להבין את ההבדלים נסביר תחילה מהו סקופ? טווח החיים של משתנה או במילים אחרות, שורות הקוד שבהן הוא מוגדר וניתן להשתמש בו.

ישנם 3 הבדלים עיקריים:

- ניתן להצהיר על אותו משתנה LET פעם אחת בלבד בתוך קוד בלוק לאחר הסקופ התוכנית לא מכירה בו ולכן ניתן להצהיר עליו פעם נוספת מחוץ לאותו סקופ, לעומת VAR שניתן להצהיר עליו כמה פעמים באותו סקופ.
- Scope - קוד בלוק שבהם משתני let או var מוגדרים: **let** הסקופ בלוקי ומתחיל מרגע שהגדרנו עליו ועד סיום הסקופ סוגריים מסולסלים שבו הוגדר "}. לעומתו **VAR** יש סקופ פונקציונלי או גלובלי יכול לחיות גם בבלוק שלא הוצהר, גם אם יוצהר בתוך בלוק פנימי הוא יזכר לאורך כל הפונקציה כולה.



3. **Hoisting** אחת מהתכונות של js היא **hoisting** התכונה הזאת קיימת בעוד שפות, מה שזה אומר שגם אם הכרזנו על משתנה בסוף הקוד הוא יוכר בתחילת הקוד אך ללא הערך שמוצב בתוכו ולכן הוא רק יוכר וערכו יהיה undefined. הוא נעשה במשתנה של VAR אך לא במשתנה LET שבו ישר תופיעה שגיאה.

לסיכום:

let הוא הכלי המועדף להצהרת משתנים ב-JavaScript המודרני, והוא מספק שליטה טובה יותר על scope ומפחית את הסיכוי לטעויות.

var	let	תכונה
ניתן להצהיר כמה פעמים	ניתן להצהיר פעם אחת	redeclare variables
פונקציונלי או גלובלי	בלוקי	Scope
כן	לא	Hoisting
ES5	ES6	מתי הוגדר
לא מומלץ בשל בעיות פוטנציאליות	מומלץ לשימוש רוב הזמן	מומלץ לשימוש

משתנים פרימיטיביים ולא פרימיטיביים:

ב-JavaScript, משתנים הם כלי לאחסון של נתונים. ישנם שני סוגים עיקריים של משתנים: פרימיטיביים ולא פרימיטיביים. ההבדל העיקרי ביניהם טמון באופן שבו הם מאוחסנים בזיכרון ובדרך שבה מתבצעות עליהם פעולות.

משתנים פרימיטיביים (By Value)

אלה משתנים שהם "קלים" ואינם תופסים מקום גדול בזיכרון אם ניקח משתנה שנקרא לו name ונשים בתוכו את הערך "maor" כעת הוא יתפוס תא בזיכרון עם הערך maor. במידה ונחליף את המשתנה ונשים בו ערך אחר לדוגמא נצהיר שמעתה name = "gross" מה שיקרא זה משתנה name יצור מקום חדש בתא אחר בזיכרון שבתוכו הערך gross מעתה לא תיהיה לנו יותר גישה לערך הקודם maor ובמידה ונעשה שימוש במשתנה name הוא תמיד יביא לנו את הערך האחרון שהוא שמר. מכאן נובע השם של משתנים פרימיטיביים **by value** למעשה ע"י הערך האחרון שהצהרנו עליו.

מאוחסנים ישירות בזיכרון: כל ערך פרימיטיבי תופס מקום קבוע וקטן בזיכרון.

ערכים בלתי משתנים: לאחר שהערך מוקצה למשתנה, הוא אינו יכול להשתנות בתוך המשתנה עצמו.

טיפוסים בסיסיים: מייצגים ערכים פשוטים כמו מספרים, מחרוזות, בוליאנים, null ו-undefined.

דוגמאות לטיפוסים פרימיטיביים:

Number: מייצג מספרים שלמים ושברים (למשל: 42, 3.14).



String: מייצג רצף של תווים (למשל: "Hello, world!")

Boolean: מייצג ערך בוליאני (אמת או שקר) (למשל: true, false)

null: מייצג ערך חסר (למשל: let x = null)

undefined: מייצג משתנה שהוכרז אך לא הוקצה לו ערך (למשל: let y);

משתנים לא פרימיטיביים (By Reference)

אלה משתנים שתופסים מקום גדול יותר בזיכרון ולכן ברגע שנשנה מבנה קודם של משתנה מהסוג הזה המערכת תפנה אותנו לאותו תא בזיכרון בו שמור המשתנה עם כל הערכים והמאפיינים שלו למעשה שינוי במאפיינים או ערכים **יפנה** אותנו אל המקור ויעדכן אותו (ידרוס את הקיים). ומכאן נובע השם **By Reference** כלומר ע"י הפניה למקור.

הפניות לאובייקטים: משתנים לא פרימיטיביים אינם מאחסנים את הערך עצמו, אלא הפניה למיקום בזיכרון שבו האובייקט מאוחסן.

ערכים ניתנים לשינוי: ניתן לשנות את התכונות והערכים של אובייקט לאחר יצירתו.

טיפוסים מורכבים: מייצגים מבנים נתונים מורכבים יותר כמו מערכים, אובייקטים, פונקציות.

דוגמאות לטיפוסים לא פרימיטיביים:

Array: אוסף סדור של ערכים (למשל: let arr = [1, 2, 3])

Object: אוסף לא סדור של זוגות מפתח-ערך (למשל: let person = { name: "John", age: 30 })

Function: מייצגת פונקציה (למשל: function greet(name) { ... })

למה חשוב להבין את ההבדל?

פעולות על משתנים פרימיטיביים ולא פרימיטיביים מתנהגות בצורה שונה. למשל, כאשר אתה משווה שני משתנים פרימיטיביים, אתה משווה את הערכים עצמם. כאשר אתה **התנהגות שונה**: משווה שני אובייקטים, אתה משווה את ההפניות שלהם, כלומר, האם הם מצביעים על אותו אובייקט בזיכרון.

העברה בפונקציות: כאשר אתה מעביר משתנה לפונקציה, משתנים פרימיטיביים מועברים כעותק, ואילו משתנים לא פרימיטיביים מועברים כהפניה.

ניהול זיכרון: הבנת ההבדל עוזרת בניהול יעיל של הזיכרון ביישומים שלך.

הבנת אופן האחסון של משתנים ב-JavaScript: מעבר על מושגים בסיסיים

שמירה ראשונית: כשאתה כותב let x = 2, אתה מבקש מהמנוע של JavaScript להקצות מקום בזיכרון כדי לאחסן את הערך המספרי 2. המקום הזה מקבל את השם "x".

שינוי הערך: כשאתה כותב x = 3, אתה לא מקצה מקום חדש בזיכרון. מה שקורה הוא שהערך בתוך המקום הקיים, שנקרא "x", משתנה מ-2 ל-3. כלומר, אתה פשוט כותב ערך חדש על הערך הקודם באותו המקום בזיכרון.

חשוב לזכור: משתנים פרימיטיביים כמו מספרים הם בלתי ניתנים לשינוי (immutable).



זה אומר שאם אתה משנה את ערכו של משתנה פרימיטיבי, אתה למעשה יוצר ערך חדש ומצביע על הערך החדש.

פונקציות הן ערכים בפני עצמם. זה אומר שאתה יכול לשמור אותן בתוך משתנים, להעביר אותן כארגומנטים לפונקציות אחרות ולהחזיר אותן כתוצאה מפעולה של פונקציה.

שינוי התנהגות הפונקציה: כשאתה משנה את התנהגות הפונקציה שמאוחסנת בתוך המשתנה Y, אתה למעשה משנה את הערך של המשתנה Y. זה לא יוצר פונקציה חדשה, אלא משנה את הקוד הפנימי של הפונקציה הקיימת.

הקשר: חשוב להבין את ההבדל בין המשתנה Y לבין הפונקציה עצמה. המשתנה Y הוא פשוט שם שמצביע על מקום בזיכרון שבו מאוחסנת הפונקציה. כשאתה משנה את הפונקציה, אתה משנה את הערך שמאוחסן בתוך המשתנה Y, אבל המשתנה עצמו נשאר אותו הדבר.

סיכום

משתנים פרימיטיביים: מאוחסנים ישירות בזיכרון ואינם ניתנים לשינוי. שינוי הערך יוצר ערך חדש.

משתנים לא פרימיטיביים (כמו פונקציות): מאחסנים הפניה לאובייקט. שינוי הערך משנה את האובייקט עצמו.

דוגמה ממחישה:

```
let x = 2; // x מצביע על הערך 2 בזיכרון
```

```
x = 3; // x מצביע כעת על הערך 3 (הערך הקודם נזנח)
```

```
let x = 2; // מצביע על הערך 2 בזיכרון x
x = 3; // מצביע כעת על הערך 3 (הערך הקודם נזנח) x

function greet(name) {
  console.log("Hello, " + name + "!");
}

let y = greet; // y מצביע על הפונקציה greet
y("John"); // יוצא: Hello, John!

// שינוי התנהגות הפונקציה
y = function(name) {
  console.log("Hi, " + name + ".");
};

y("Alice"); // יוצא: Hi, Alice.
```

לסיכום:



כשאתה משנה את ערכו של משתנה פרימיטיבי, אתה יוצר ערך חדש ומצביע על הערך החדש.

כשאתה משנה את ערכו של משתנה לא פרימיטיבי (כמו פונקציה), אתה משנה את האובייקט עצמו, ולא יוצר אובייקט חדש.

By Value ו-By Reference ב-JavaScript: הבנה עמוקה

המושגים **By Value** ו-**By Reference** הם יסודיים להבנת אופן העברת נתונים בין משתנים ופונקציות ב-JavaScript. הם קשורים ישירות לאופן שבו JavaScript מטפל בסוגי נתונים שונים.

By Value: העברה של עותק

מה זה: כאשר משתנה מועבר ב-By Value, למעשה מועבר עותק של הערך שלו. כל שינוי שעושים בתוך הפונקציה לא ישפיע על הערך המקורי של המשתנה מחוץ לפונקציה.

מי משתמש בזה: בעיקר טיפוסים פרימיטיביים כמו מספרים, מחרוזות, בוליאנים, null ו-undefined.

```
let x = 5;
function changeValue(num) {
  num = 10;
}
changeValue(x);
console.log(x); // יפיק 5
```

*הסבר: למרות שבתוך הפונקציה changeValue שינו את הערך של num ל-10, השינוי הזה לא השפיע על הערך המקורי של x מחוץ לפונקציה, מכיוון שהועבר עותק של הערך.

By Reference: העברה של הפניה

מה זה: כאשר משתנה מועבר ב-By Reference, למעשה מועברת הפניה למיקום בזיכרון שבו מאוחסן הערך. כל שינוי שעושים בתוך הפונקציה ישפיע ישירות על הערך המקורי של המשתנה מחוץ לפונקציה.

מי משתמש בזה: בעיקר טיפוסים מורכבים כמו אובייקטים, מערכים ופונקציות.

דוגמה:

```
let person = { name: "John" };
function changeName(obj) {
  obj.name = "Alice";
}
changeName(person);
console.log(person.name); // יפיק Alice
```

*הסבר: בתוך הפונקציה changeName שינו את שם האובייקט person, והשינוי הזה השפיע ישירות על האובייקט המקורי, מכיוון שהועברה הפניה לאובייקט.

למה זה חשוב?



הבנת התנהגות הקוד: הבנת ההבדל בין By Value ו-By Reference חיונית כדי לחזות כיצד הקוד שלך יתנהג.

מניעת באגים: אי הבנה של המושגים האלה יכולה להוביל לתוצאות בלתי צפויות ולבאגים קשים לאיתור.

כתיבת קוד יעיל: הבנה זו מאפשרת לך לכתוב קוד יעיל יותר ולנצל בצורה טובה יותר את המשאבים של JavaScript.

לסיכום:

By Value: העברה של עותק, שינויים לא משפיעים על המקור.

By Reference: העברה של הפניה, שינויים משפיעים על המקור.

-Prompt: תיבת קלט המקבלת ערך, במידה ונשמור את הפונקציה prompt בתוך משתנה, המשתנה יקבל את הערך שהמשתמש הזין בהודעה שקפצה לו על המסך.

אופרטורים

הם סימנים המאפשרים לנו לבצע פעולות במשתנים או ערכים, הם מהווים את הבסיס לחישובים השוואות והיגיון בתוכניות java script. למשל: +, -, *, /, =, <, >, !, ?. .

אופרטור Spread - מגדירים על ידי 3 נקודות ...

למעשה משכפל מערך או אובייקט קיים ללא קשר ישיר אליו זאת אומרת אם נבצע שינויים במערך החדש זה לא יגרור שינויים למערך המקורי.

ספרייד אופרטור מפרק איברים ומעתיק אותם למערך חדש בצורת מחרוזת עם פסיקים. במידה ונרצה לפרק אובייקט ולהעתיק אותו לאובייקט חדש נפתח תחילה סוגריים מסולסלים של אובייקט ובתוכן נבצע ספרייד לאותו אובייקט שאנחנו רוצים להעתיק כמו כן לגבי מערכים, נפתח תחילה סוגריים מרובעים ובתוכן נבצע ספרייד למערך שאנחנו רוצים להעתיק. בדוגמא הבאה נראה כיצד עלינו לבצע העתקה של מערכים ואובייקטים.

```
let obj = {id: "01", name: "maor"};
let array = [1,2,3];

myObjCopy = {...obj};
myArrayCopy = [...array];
```

בדוגמא הבאה ניקח 2 מערכים שונים ונחבר אותם לכדי מערך אחד חדש:

```
let numbers1 = [1, 2, 3];
let numbers2 = [4, 5, 6];
let numbers = [...numbers1, ...numbers2];
```

יצירת עותקים - copy



Simple copy – משנה את המקור

```
let user = {
  id: "01",
  name: "maor",
}

let user2 = user;
```

מפנה אל המקור, כאשר נשנה את user2 (שכפול) גם האובייקט user (מקור) ישתנה בהתאם.

Shallow copy – עותק חלקי

אם יש לנו אובייקט או מערך פנימי הוא עדיין יקושר אל המקור.

```
//אובייקט שבתועו אובייקט פנימי
let person = {
  name: "maor",
  age: 34,
  adress: {
    city: "megadim",
    country: "israel"
  }
}

let person2 = { ...person };

//ישתנה רק את העותק
person2.name = "haim";
//ישנה גם את המקור
person2.adress.city = "haifa";
```

Deep Copy – עותק אמיתי לכל חלקיו

```
//שכפול אובייקטים פנימי וחיצוני
let person3 = { ...person, adress: { ...person.adress } };
//ישתנה רק ההעתק
person3.name = "moshe";
person3.adress.city = "TLV";
console.log("העתק", person3);
```

כל פעם שנרצה להעתיק מערכים או אובייקטים שבתוכם קיימים מערכים או אובייקטים פנימיים נצטרך לגשת לכל מערך או אובייקט בנוסף על מנת לוודא העתק מוחלט.

בדוגמה הנ"ל כאשר ביצענו ספרייד לאובייקט PERSON יכולנו לאחר מכן לגשת למפתח ADDRESS מכיוון שהמערכת מכירה אותו ולהעתיק גם אותו בנוסף, התוצאה העתק של אובייקט המקור לאובייקט חדש במידה ונבצע בו שינויים או באובייקט הפנימי שלו ADDRESS השינויים יתרחשו רק על האובייקט החדש ללא שינוי באובייקט המקורי.



- ניתן להשתמש בספרייט אופרטור גם לצורך פירוק מפתחות של אובייקטים או חילוץ איברים של מערך למשל לארגיומטים של פונקציה:

```
• let numbers = [1,2,3];  
• function sum (a,b,c) {  
•   return a+b+c;  
• }  
•  
• //דוגמא זו אנו שולחים את האיברים במערך כארגיומטים לפונקציה  
• console.log (sum(...numbers));  
• //דפיס 6
```

פרמטרים

ניתן לדמיין מתכנון לבישול המכיל רכיבים שימשו אותנו להכנת המנה. פרמטרים אלו שמות שאנו נותנים לערכים שפונקציה מצפה לקבל את הפרמטרים נכניס תחילה בתוך הסוגריים של הפונקציה ונשתמש בהם בתוך הפונקציה כדי לבצע את המשימה שלה. ניתן להגדיר גם פרמטר דיפולטיבי שיש לו ערך ברירת מחדל, רק כאשר נקבל ערך undefined הוא יקח באופן דיפולטיבי את הערך שהגדרנו מראש בתוך הסוגריים למשל:

```
// פרמטרים  
function sum(num1, num2 = 10) {  
  let result = num1 + num2;  
  console.log(result);  
}
```

ארגיומנטים

אלו הערכים האמיתיים שאנו מזינים לפונקציה כאשר אנו קוראים לה הם ממלאים את המקומות השמורים שהוגדרו כפרמטרים. הם מופיעים בתוך הסוגריים כאשר אנו קוראים לפונקציה למשל:

```
<!-- ארגיומנטים -->  
<button onclick="result(1,2)"></button>
```

שימוש במילה this

המילה השמורה this ב-JavaScript היא אחת המושגים החשובים ביותר, אך גם המבלבלים ביותר, בשפה. היא מתייחסת לאובייקט הנוכחי, אך הערך המדויק שלה משתנה בהתאם להקשר שבו היא משמשת.

מה זה this בעצם?

בכל פונקציה ב-JavaScript, this מצביע על אובייקט מסוים. האובייקט הזה נקרא "הקשר" (context) של הפונקציה. הערך של this נקבע בזמן ההפעלה של הפונקציה, ולא בזמן ההגדרה שלה.



מתי משתמשים ב-this?

בתוך אובייקט: כאשר this משמשת בתוך מתודה של אובייקט, היא מצביעה על האובייקט עצמו. זה מאפשר לגשת לתכונות ולמתודות אחרות של האובייקט.

בפונקציות רגילות: כאשר this משמשת בתוך פונקציה רגילה (לא מתודה של אובייקט), הערך שלה תלוי באיך הפונקציה הוזמנה. בדרך כלל, היא מצביעה על האובייקט הגלובלי (window ברוב הדפדפנים).

בפונקציות מחוברות (bound functions): ניתן להשתמש בשיטות כמו bind, call ו-apply כדי לשנות את הערך של this בתוך פונקציה. זה שימושי כאשר רוצים להעביר פונקציה כארגומנט, אך רוצים שהיא תפעל בהקשר מסוים.

דוגמאות:

```
// 1 דוגמה: this בתוך אובייקט
const person = {
  firstName: "John",
  lastName: "Doe",
  fullName: function() {
    return this.firstName + " " + this.lastName;
  }
};

console.log(person.fullName()); // יפיק: John Doe

// 2 דוגמה: this בפונקציה רגילה
function sayHello() {
  console.log("Hello, " + this.name);
}

sayHello(); // יפיק: Hello, undefined (באובייקט הגלובלי name אם אין תכונה)

// 3 דוגמה: bind-שימוש ב
const person1 = { name: "Alice" };
const person2 = { name: "Bob" };

const sayHelloBoundToPerson1 = sayHello.bind(person1);
sayHelloBoundToPerson1(); // יפיק: Hello, Alice
```

מקרים מיוחדים

הקשר של this בתוך חצים (arrow functions): בתוך חצים, this תמיד שומר על הערך של this מההקשר החיצוני.

הקשר של this באירועים: כאשר מטפלים באירועים, הערך של this בתוך מטפל האירוע יכול להשתנות בהתאם להגדרות האירוע.



למה כל כך מבלבל?

הערך של `this` דינמי: הוא משתנה בהתאם להקשר.

תלות בהקשר ההפעלה: האופן שבו פונקציה הוזמנה משפיע על הערך של `this` בתוכה.

שימוש שונה בשפות תכנות אחרות: ההתנהגות של `this` ב-JavaScript שונה משפות אחרות.

סיכום

הבנת `this` היא קריטית לתכנות אובייקטים ב-JavaScript. היא מאפשרת לנו ליצור קוד מודולרי וגמיש יותר. למרות המורכבות שלה, עם תרגול והבנה עמוקה של ההקשרים השונים, ניתן להשתמש ב-`this` בצורה יעילה וטבעית.

פונקציות

פונקציות הם כמו מתכונים קטנים של קוד. הם מבצעות פעולה ספציפית ומאפשרות לנו לארגן את הקוד בצורה מודולרית הניתנת לשימוש חוזר. פונקציות מקבלות ומחזירות ערכים. סוגי פונקציות: פונקציה רגילה, פונקציה ביטוינית ניתן לשים אותה בתוך משתנה למשל:

```
let firstName = "maor";
let LastName = "gross";
const fullName = function userName(firstName, LastName) {
  return "wellcome " + firstName + " " + LastName;
};

console.log(fullName(firstName, LastName));

// wellcome maor gross
```

פונקציית חץ - arrow function

פונקציות חץ (Arrow functions) הן דרך קצרה וקלה יותר להגדיר פונקציות ב-JavaScript. הן הוצגו ב-ES6 והן הפכו לחלק בלתי נפרד מסגנון הכתיבה המודרני של JavaScript.

היתרונות של פונקציות חץ:

סינטקס קצר יותר: הסינטקס של פונקציות חץ הוא פשוט וקומפקטי יותר מאשר הסינטקס של פונקציות רגילות.

קשר ל-`this`: פונקציות חץ לא יוצרות את הקשר ה-`this` שלהן עצמן, אלא יורשות אותו מההקשר שבו הן מוגדרות. זה יכול להקל על עבודה עם `this` בתוך פונקציות.

לפונקציות קטנות: פונקציות חץ מצוינות לפונקציות קטנות וחד פעמיות.

מבנה בסיסי של פונקציית חץ:

```
(פרמטרים) => {
  // הפונקציה
}
```



```
};
```

פרמטרים: רשימת הפרמטרים של הפונקציה, כמו בפונקציות רגילות.

<=: סימן החץ המחובר בין הפרמטרים לגוף הפונקציה.

גוף הפונקציה: הקוד שיבוצע כאשר הפונקציה תקרא.

כללים:

1. במידה ויש לנו רק פרמטר אחד ניתן לוותר על הסוגריים המעוגלים של הפרמטרים לדוגמא:

```
sum => {  
  return num1+num2;  
}
```

2. במידה ואין פרמטרים בכלל יש לכתוב סוגריים לדוגמא:

```
() => {  
  return num1+num2;  
}
```

3. במידה והפונקציה מחזירה ערך מיד, נוותר על הסוגריים המסולסלים ועל המילה return לדוגמא:

```
() => num1+num2;
```

4. במידה והפונקציה לא מחזירה ערך מיד נשתמש בסוגריים מסולסלים ובמילה return לפי הצורך (ניתן יהיה להשתמש במשתנה ובערך שלו מחוץ לפונקציה בזכות המילה return לדוגמא:

```
5. (num1, num2) => {  
6.   let sum = num1 + num2;  
7.   let avg = sum / 2;  
8.   return avg;  
9. }  
10.  
11. console.log(avg);
```

מתי כדאי להשתמש בפונקציות חץ?

כאשר אתה צריך פונקציה קטנה וחד פעמית.

כאשר אתה עובד עם מערכים ושיטות כמו map, filter, reduce וכו'.

כאשר אתה רוצה לקשור את ה-this להקשר הנוכחי.

מתי כדאי להימנע משימוש בפונקציות חץ?



כאשר אתה צריך פונקציה גדולה ומורכבת.

כאשר אתה צריך גמישות רבה יותר בקשר ה-`this`.

כאשר אתה צריך להגדיר פונקציה כ-`constructor`.

לסיכום:

פונקציות חץ הן כלי רב עוצמה ב-`JavaScript`. הן מאפשרות לך לכתוב קוד קצר יותר ונקי יותר. עם זאת, חשוב להשתמש בהן בצורה נכונה כדי להפיק את המיטב מהן.

מהי הפונקציה `Math.random()`?

הפונקציה `Math.random()` ב-`JavaScript` היא כלי בסיסי ליצירת מספרים אקראיים. כאשר קוראים לפונקציה זו, היא מחזירה מספר עשרוני אקראי בין 0 (כולל) ל-1 (בלעדי). כלומר, התוצאה יכולה להיות 0, אבל לעולם לא תהיה 1.

איך היא עובדת?

המחשב אינו יכול לייצר מספרים אקראיים אמיתיים, אלא משתמש באלגוריתמים שמייצרים רצף של מספרים שנראים אקראיים. אלגוריתמים אלו נקראים "מחוללי מספרים פסאודו-אקראיים".

דוגמאות:

הטלת קובייה: `Math.floor(Math.random() * 6) + 1`

בחירה אקראית בין 0 ל-100: `Math.floor(Math.random() * 101)`

יצירת צבע אקראי: `(Math.random()*0xFFFFFF<<0).toString+'#'` (16)

מהי הפונקציה `Math.floor()`?

הפונקציה `Math.floor()` ב-`JavaScript` משמשת לעיגול מספר למספר השלם הקטן ביותר ששווה לו או קטן ממנו. במילים אחרות, היא מעגלת כל מספר למספר השלם הקרוב ביותר כלפי מטה.

איך היא עובדת?

קלט: הפונקציה מקבלת מספר כפרמטר.

פלט: היא מחזירה את המספר השלם הגדול ביותר ששווה או קטן מהמספר שהועבר לה.

דוגמאות:

`Math.floor(3.14)` יחזיר 3.

`Math.floor(9.99)` יחזיר 9.

`Math.floor(-2.7)` יחזיר -3. (שימו לב שעבור מספרים שליליים, העיגול ילך לכיוון המספר השלם הקטן יותר במספרים השליליים)

`Math.floor(0)` יחזיר 0.

מתי משתמשים ב-`Math.floor()`?



יצירת מספרים שלמים אקראיים: בשילוב עם `Math.random()`, ניתן ליצור מספרים שלמים אקראיים בטווח מסוים.

עיבוד נתונים מספריים: לעיתים יש צורך לעבוד עם מספרים שלמים בלבד, ו-`Math.floor()` מאפשרת להסיר את החלק העשרוני של מספר.

אינדקסים במערכים: כאשר עובדים עם מערכים, האינדקסים חייבים להיות מספרים שלמים. `Math.floor()` יכולה לשמש לוודא שהאינדקס תקין.

```
// יצירת מספר שלם אקראי בין 1 ל-10
const randomInteger = Math.floor(Math.random() * 10) + 1;
console.log(randomInteger);
```

סיכום:

הפונקציה `Math.floor()` היא כלי שימושי מאוד ב-JavaScript כאשר רוצים לעבוד עם מספרים שלמים. היא מספקת דרך פשוטה ויעילה לעגל מספר למספר השלם הקטן ביותר.

מה עושה הפונקציה `math.ceil`?

הפונקציה `math.ceil()` היא פונקציה מתמטית שנמצאת בשפות תכנות רבות, ביניהן JavaScript. תפקידה הוא לעגל מספר כלפי מעלה למספר השלם הקרוב ביותר. במילים אחרות, היא תיקח כל מספר עשרוני ותחזיר את המספר השלם הבא אחריו.

דוגמאות:

`math.ceil(3.14)` יחזיר 4.

`math.ceil(9.99)` יחזיר 10.

`math.ceil(2.5)` יחזיר 3 (שיים לב: עיגול כלפי מעלה עבור מספר שלילי פירושו להתקרב לאפס).

מתי משתמשים בפונקציה הזו?

עיגול מחירים: אם אתה בונה חנות מקוונת ואתה רוצה לעגל את המחירים כלפי מעלה לסכום השלם הבא, `math.ceil()` היא הפונקציה המתאימה.

חישובים מתמטיים: בפעולות מתמטיות שונות, לעיתים יש צורך לעגל תוצאה כלפי מעלה כדי לקבל מספר שלם.

עבודה עם מערכים: אם אתה עובד עם מערכים בגדלים משתנים, `math.ceil()` יכולה לעזור לך לחשב את הגודל המינימלי הנדרש.

שימושים נוספים:

עיגול זמן: אם אתה רוצה לעגל שניות לדקות או דקות לשעות, `math.ceil()` יכולה להיות שימושית.

חישוב מספר העמודים: אם אתה יודע את מספר המילים במאמר ואת מספר המילים הממוצע בעמוד, `math.ceil()` יכולה לעזור לך לחשב את מספר העמודים המינימלי הנדרש.

סיכום:



הפונקציה `math.ceil()` היא כלי שימושי מאוד בכל הקשור לעיגול מספרים כלפי מעלה. היא פשוטה לשימוש ומאפשרת לך לבצע חישובים מדויקים יותר.

פונקצית CALL BACK

פונקצית קריאה חוזרת אשר קוראת לעצמה או מועברת כארגומנט לפונקציה אחרת מאפשרת לבצע פעולות אסינכרוניות בקוד להמתין לביצוע לאחר שהפעולה תסתיים היא תקרא לפונקציה `callback` כדי לטפל בתוצאה, במילים פשוטות זו דרך להפעיל פונקציה אחת מתוך פונקציה אחרת. לדוגמא:

```
document.getElementById("myButton").addEventListener("click", function  
( ) {  
    alert("Hello World");  
});
```

בעת לחיצה על הכפתור, הפונקציה `addEventListener` תקרא לפונקציה אנונימית `callback` שיצרנו שמחזירה הודעה `Hello World`. בואו נראה דוגמא נוספת:

```
document.getElementById("myButton").addEventListener("click", sayHola);
```

הפונקציה `addEventListener` מצפה לקבל סוג אירוע+פונקציה, לכן אין צורך להוסיף סוגריים עגולים לאחר הפונקציה `sayHola` במידה ונוסיף סוגריים עגולים התוכנית תקרא לה מיד במידה ולא נוסיף סוגרים אלא רק את שם הפונקציה התוכנית תקרא לה בתגובה לאירוע שהגדרנו. בואו נראה דוגמא לפונקציה שמקבלת פונקציה כפרמטר:

```
function sayThankYou ( ) {  
    alert ("Thank You");  
}  
  
//פונקציה דינמית, מקבלת פונקציה כפרמטר  
function calcTotalPrice (price,qproducts,myFunc) {  
    alert (price*qproducts);  
    //קוראת לפונקציה שקיבלה  
    myFunc();  
}  
  
//פונקציה שמקבלת פונקציה כארגומנט  
calcTotalPrice(5,10,sayThankYou);
```

כעת נראה דוגמא לפונקציה שמקבלת פונקציה שיש לה פרמטר:

```
// Callback function + parameters  
function cartTotal(productPrice, quantity, myNewFunc) {  
    let total = productPrice * quantity;  
    // מפעילה פונקציה עם פרמטר  
    return myNewFunc(total);  
}
```



```
}  
//פונקציה מקבלת פרמטר/  
function orderSuccess(totalPrice) {  
  //מדפיסה את הפרמטר/  
  console.log(`Thank You. Total price: ${totalPrice}`);  
}  
  
//המעלת פונקציה שמקבלת כאגרוימט פונקציה נוספת/  
cartTotal(20, 7, orderSuccess);
```

כעת נראה דוגמא לפונקציה שמחזירה ערך:

```
function myCalc () {  
  let total = 5+10;  
  return total;  
};
```

הפונקציה myCalc מחזירה את total באמצעות המילה return. בעת ביצוע תרגילים או משימות כאשר יגידו לנו הפונקציה מקבלת הכוונה תהיה לפרמטר וכאשר יגידו לנו הפונקציה מחזירה הכוונה לערך שהפונקציה מחזירה באמצעות return.

בואו נראה דוגמא לפונקציה מופעלת ברקע:

```
function first(){  
  setTimeout(() => {  
    console.log("im first");  
  }, 3000);  
}  
  
function last(){  
  console.log("Im last")  
}  
  
first(); // A  
last(); // B
```

2 הפונקציות first ו-last מופעלות אך:

B יודפס מייד לפני A

A יודפס לאחר 3 שניות.

דוגמא נוספת לפונקציה שמקבלת פונקציה כפרמטר:

```
// Callback  
// A
```




```
let printFirst = (callbackFunc) => {
  setTimeout(() => {
    console.log("Login Success");
    callbackFunc();
  }, 3000);
}

//B
let printLast = () => {
  console.log("Redirecting to HomePage");
}

printFirst(printLast);
```

פונקציית קריאה חוזרת היא פונקציה שמועברת כארגומנט לפונקציה אחרת, ומבוצעת לאחר שהפונקציה הראשונה מסיימת את פעולתה. במילים אחרות, זהו סוג של "הוראה" לפונקציה הראשונה: "כשתסיימי את העבודה שלך, תקראי לפונקציה הזאת".

למה משתמשים בפונקציות קריאה חוזרות?

אסינכרוניות: פונקציות קריאה חוזרות משמשות רבות בעולם האסינכרוני של JavaScript, למשל בעבודה עם רשת, טיימרים, או פעולות שעלולות לקחת זמן. זה מאפשר לקוד שלך להמשיך להתבצע בזמן שהפעולה האסינכרונית מתבצעת ברקע, ולקבל התראה כאשר היא מסתיימת.

עיצוב קוד: פונקציות קריאה חוזרות יכולות לשפר את קריאות הקוד, להפריד בין חלקים שונים של הקוד, ולעזור ליצור קוד מודולרי יותר. בנוסף מאפשרת לנו לנהל אירועים ופעולות שרשרת ועושה לנו סדר בקוד.

דוגמאות:

נניח שאנחנו רוצים להציג הודעה לאחר המתנה של 2 שניות. נוכל להשתמש בפונקציית `setTimeout` שמקבלת פונקציה כארגומנט:

```
function greet(name) {
  console.log(`Hello, ${name}!`);
}

setTimeout(greet, 2000, "World");
```

```
function loop() {
  console.log("אני חוזר על עצמי");
  loop(); // קוראת לעצמה שוב
}

loop(); // מפעילה את הלולאה בפעם הראשונה
```



```
function loop(count) {  
  console.log("אני חוזר על עצמי: " + count);  
  if (count < 5) {  
    loop(count + 1);  
  }  
}  
  
loop(1);
```

הפונקציה trim() ב-JavaScript:

מה עושה הפונקציה trim()?

הפונקציה trim() ב-JavaScript משמשת להסרת רווחים לבנים (whitespace) מראש ומסוף מחרוזת. רווחים לבנים כוללים רווחים רגילים, תווי טאב, ושורות חדשות. הפונקציה הזו שימושית מאוד כאשר אתה עובד עם נתונים טקסטואליים שמקורם למשל מטפסים, משתמשים או ממקורות אחרים שעלולים להכיל רווחים מיותרים.

איך היא עובדת?

קבלת מחרוזת: הפונקציה מקבלת מחרוזת כקלט.

הסרת רווחים: היא סורקת את המחרוזת משני הקצוות ומסירה את כל התווים הלבנים עד שהיא מגיעה לתו שאינו לבן.

החזרת מחרוזת חדשה: הפונקציה מחזירה מחרוזת חדשה שבה הוסרו כל הרווחים הלבנים מהקצוות, מבלי לשנות את המחרוזת המקורית.

```
const myString = " Hello, World! ";  
const trimmedString = myString.trim();  
console.log(myString.trim()); // יפיק: "Hello, World!"
```

הפונקציה toFixed

פונקציית toFixed() ב-JavaScript משמשת לעיצוב מספר בצורה של מחרוזת, עם מספר קבוע של ספרות אחרי הנקודה העשרונית. במילים אחרות, היא מעגלת מספר ומחזירה מחרוזת המייצגת את המספר המעוגל עם מספר מוגדר של ספרות אחרי הנקודה.

איך משתמשים בה?

הסינטקס הבסיסי של הפונקציה הוא:

```
number.toFixed(digits);
```

number: המספר אותו רוצים לעגל ולעצב.



digits: מספר הספרות אחרי הנקודה העשרונית שברצוננו שיופיעו במחרוזת התוצאה.

הפונקציה toUpperCase

פונקציית toUpperCase () ב-JavaScript היא שיטה (method) שמשמשת על מחרוזות (strings). תפקידה הוא להמיר את כל האותיות הקטנות במחרוזת לאותיות גדולות. במילים אחרות, היא "מעלה" את כל האותיות באנגלית במחרוזת לאותיות רישיות.

איך משתמשים בה?

הסינטקס הבסיסי של הפונקציה הוא:

```
string.toUpperCase();
```

string: המחרוזת עליה רוצים להפעיל את הפונקציה.

```
let text = "hello world";  
let uppercaseText = text.toUpperCase();  
console.log(uppercaseText); // יפיק: HELLO WORLD
```

נקודות חשובות לזכור:

החזרת מחרוזת חדשה: הפונקציה toUpperCase () מחזירה מחרוזת חדשה, ולא משנה את המחרוזת המקורית. מחרוזות ב-JavaScript הן אי-משתנות (immutable), כלומר אי אפשר לשנות אותן בודדת במחרוזת קיימת.

רק אותיות אנגליות: הפונקציה משפיעה רק על אותיות האלפבית האנגלי. תווים אחרים, כמו מספרים, סימני פיסוק ותווים בשפות אחרות, יישארו ללא שינוי.

תמיכה בדפדפנים: הפונקציה toUpperCase () נתמכת באופן נרחב בכל הדפדפנים המודרניים.

מתי משתמשים ב-toUpperCase()?

הצגת טקסט באופן אחיד: כאשר רוצים להציג טקסט באופן אחיד, למשל ככותרות או כתוביות.

חיפוש טקסט: כאשר רוצים לבצע חיפוש טקסט ולא חשוב אם המילים נכתבו באותיות גדולות או קטנות.

עיבוד טקסט: הפונקציה יכולה להיות שימושית כחלק מעיבוד טקסט מורכב יותר.

מערכים - הכרות + מתודות

Push() – מוסיף לאיבר האחרון במערך

Pop() – מסיר את האיבר האחרון במערך



Unshift – מוסיף איבר לאינדקס הראשון במערך

Shift() – מסיר את האיבר הראשון במערך

splice – הפונקציה splice() היא כלי רב עוצמה ב-JavaScript המאפשר לך לערוך מערכים באופן דינמי. היא מאפשרת לך:

להסיר: להסיר איברים ממערך החל מאינדקס מסוים.

להוסיף: להוסיף איברים חדשים למערך באינדקס מסוים.

להחליף: להחליף איברים קיימים באיברים חדשים.

איך היא עובדת?

הפונקציה splice() מקבלת מספר פרמטרים:

index: האינדקס שבו ברצונך להתחיל את השינוי.

deleteCount: מספר האיברים שברצונך להסיר (אופציונלי).

item1, item2, ...: רשימת האיברים שברצונך להוסיף (אופציונלי).

דוגמה:

נניח שיש לנו מערך של פירות:

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
```

כדי להסיר את הפרי "Orange" ולהוסיף את הפירות "Kiwi" ו-"Pineapple", נוכל להשתמש ב-splice() כך:

```
fruits.splice(1, 1, "Kiwi", "Pineapple");
```

1: האינדקס של "Orange".

1: מספר האיברים להסרה (1, כלומר "Orange").

"Kiwi", "Pineapple": האיברים להוספה.

אחרי השינוי, המערך fruits יהיה:

```
["Banana", "Kiwi", "Pineapple", "Apple", "Mango"];
```

דוגמאות נוספות:

מחיקת כל האיברים מאינדקס 2 ומעלה.

```
fruits.splice(2);
```



הוספת שלושה איברים חדשים בתחילת המערך:

```
fruits.splice(0, 0, "Lemon", "Grape", "Watermelon");
```

החלפת שני האיברים הראשונים במערך:

```
fruits.splice(0, 2, "Strawberry", "Blueberry");
```

חשוב לזכור:

splice() משנה את המערך המקורי. אם ברצונך לשמור את המערך המקורי, צור עותק שלו לפני השימוש ב-splice().

האינדקסים במערך מתחילים מ-0.

אם תציין אינדקס גדול יותר מאורך המערך, לא יבוצע שום שינוי.

סיכום:

הפונקציה splice() היא כלי גמיש מאוד לעבודה עם מערכים ב-JavaScript. היא מאפשרת לך לבצע מגוון רחב של פעולות על מערכים, החל משינויים פשוטים ועד למניפולציות מורכבות יותר.

מתודות

תחילה יש לציין נקודות חשובות:

- כל המתודות מקבלות פונקציה בתוך הסוגרים.
- כל המתודות עוברות על כל איברי המערך ומחזירות מערך חדש עם מניפולציה כפי שהגדרנו בפונקציה.

REDUCE

מתודה המאפשרת לנו לבצע צמצום של איברי המערך לידי פרט מידע יחיד.

המתודה reduce() היא אחת מהמתודות החזקות ביותר לעבודה עם מערכים ב-JavaScript. היא מאפשרת לנו "לקפל" מערך שלם לערך יחיד, על ידי ביצוע פעולה מצטברת על כל האיברים במערך.

איך היא עובדת?

המתודה reduce() מקבלת שתי פונקציות כארגומנטים:

פונקציית callback: פונקציה זו תקבל ארבעה ארגומנטים:

accumulator: הערך המצטבר הנוכחי (הערך ההתחלתי שאתה מגדיר או הערך הקודם שנוצר).

currentValue: הערך הנוכחי במערך שאתה מעבד.

currentIndex: האינדקס של הערך הנוכחי במערך. (אופציונלי)



array: המערך המקורי שעליו אתה מבצע את הפעולה. (אופציונלי)
הפונקציה צריכה להחזיר את הערך החדש של accumulator, שיהפוך לערך ההתחלתי עבור האיבר הבא במערך.

initialValue (אופציונלי): ערך התחלתי עבור accumulator. אם לא תציין ערך, הערך ההתחלתי יהיה האיבר הראשון מיקום [0] במערך והאיטרציה תתחיל מהאיבר השני.

דוגמאות:

חישוב סכום כל האיברים במערך:

```
const numbers = [1, 2, 3, 4];
const sum = numbers.reduce((accumulator, currentValue) => {
  return accumulator + currentValue;
});
console.log(sum); // יפיק: 10
```

מציאת הערך המקסימלי במערך:

```
const numbers = [10, 5, 8, 12];
const max = numbers.reduce((accumulator, currentValue) => {
  return Math.max(accumulator, currentValue);
});
console.log(max); // יפיק: 12
```

ספירת מספר המילים במערך של משפטים:

```
const sentences = ['hello world', 'how are you'];
const wordCount = sentences.reduce((count, sentence) => {
  return count + sentence.split(' ').length;
}, 0);
console.log(wordCount); // יפיק: 5
```

FIND

מחזירה את האיבר הראשון שעומד בתנאי מסוים.

```
let numbers = [5, 10, 20, 30, 40];
let numbersFind = numbers.find((number) => {
  return number < 20;
});
console.log(numbersFind);
// יציג את האיבר הראשון הקטן מ-20.
```

Filter



בדומה ללולאת ForEach גם הוא רץ על כל המערך ואנחנו מצפים לקבל return של true או false אם נקבל אמת הוא יעשה פוש למערך ואם נקבל שקר הוא ימשיך לערך הבא לדוגמא:

```
let numbers = [5, 10, 20, 30, 40];
let numbersFilter = numbers.filter((number) => {
  return number.includes("0");
});
// תחזיר מערך חדש עם כל האיברים המכילים את הסיפרה 0
```

דוגמאות נוספות:

```
const products_arr = [
  { name: "apple", price: 6 },
  { name: "banana", price: 10 },
  { name: "kiwi", price: 8 },
  { name: "lemon", price: 9 },
];

const init = () => {
  let filter_arr = products_arr.filter((item) => {
    if (item.price < 9) {
      return true;
    } else {
      return false;
    }
  });
  console.log(filter_arr);
};

init();
```

ניתן גם לקצר בהתבסס על מה שלמדנו בפונקציית חץ arrow function מכיוון שפילטר מחזירה ישר ערך נותר על הסוגריים ועל המילה return כמו בדוגמא הבאה:

```
const init = () => {
  let filter_arr = products_arr.filter((item) => item.price < 9);
  console.log(filter_arr);
};
```

ההדפסה שנקבל בקונסול היא מערך לאחר סינון של המערך המקורי products_arr המכיל אובייקטים שהמחיר שלהם קטן מ9 שהם למעשה apple,kiwi וזה יהיה המערך החדש בשם filter_arr:

```
filter_arr = [
  { name: "apple", price: 6 },
  { name: "kiwi", price: 8 },
];
```



מבנה הפונקציה:

```
const newArray = array.filter(function (currentValue, index, arr) {  
  // אם ברצונך לכלול את האלמנט במערך החדש true החזר  
  // אם ברצונך להשמיט את האלמנט החזר false  
});
```

array: המערך המקורי שברצונך לסנן.

Function (currentValue, index, arr): פונקציית בדיקה שמקבלת עד שלושה

ארגומנטים:

- **currentValue**: הערך הנוכחי של האלמנט במעבר הנוכחי.
- **index**: האינדקס של האלמנט הנוכחי במערך המקורי.
- **arr**: המערך המקורי עצמו.
- **newArray**: המערך החדש שנוצר, המכיל רק את האלמנטים שעברו את הבדיקה.

Map

מחזיר מערך חדש שבנוי עם מניפולציה שהגדרנו לו למשל אם יש לנו מערך מורכב שבותוכו עוד מערכים ואובייקטים כמו המערך הבא:

```
let numbers = [10,20,30,40];  
newNumbers =numbers.map((number) => {  
  return number/2;  
})  
// תחזיר מערך חדש לאחר חישוב 50% מהמספרים במערך הקודם
```

דוגמא נוספת:

```
const users = [  
  {  
    first: "maor",  
    last: "gross",  
    location: {  
      country: "IL",  
      city: "megadim",  
      street: "hahofim",  
      number: "100",  
    },  
    dob: [  
      {  
        age: 34,  
        birthday: "07/04/90",  
      },  
    ],  
  },  
];
```



```
];
```

ואנחנו רוצים לבנות מערך מערכים ספציפים שיהיו לנו קלים ליישום בתוכנית אז נגדיר את המערך מחדש באמצעות MAP כמו בדוגמא הבאה:

```
const initMap = () => {
  let map_arr = users.map((item, i) => {
    return {
      fullname: item.first + " " + item.last,
      location:
        item.location.street +
        " " +
        item.location.number +
        " " +
        item.location.city,
      age: item.dob[i].age,
    };
  });
  console.log(users);
  console.log(map_arr);
};

initMap();
```

כעת לאחר ההדפסה של שני המערכים בקונסול נוכל לראות את ההבדלים בין מערך users שנראה כך:



[app.js:133](#)

```
▼ [...] i
  ▼ 0:
    ▼ dob: Array(1)
      ▼ 0:
        age: 34
        birthday: "07/04/90"
        ► [[Prototype]]: Object
        length: 1
        ► [[Prototype]]: Array(0)
        first: "maor"
        last: "gross"
    ▼ location:
      city: "megadim"
      country: "IL"
      number: "100"
      street: "hahofim"
      ► [[Prototype]]: Object
      ► [[Prototype]]: Object
      length: 1
      ► [[Prototype]]: Array(0)
```

לבין מערך map_arr שמוצג באופן הרבה יותר ברור וקל ליישום בתוכנית ויראה כך:

[app.js:134](#)

```
▼ [...] i
  ▼ 0:
    age: 34
    fullname: "maor gross"
    location: "hahofim 100 megadim"
    ► [[Prototype]]: Object
    length: 1
    ► [[Prototype]]: Array(0)
```

המתודה sort () ב-JavaScript

משמשת למיין מערך לפי ערכיו בצורה בינארית כמו סינון A-Z.



באופן בסיסי, היא מסדרת את האלמנטים לפי סדר עולה.

אותיות גדולות קודמות לאותיות קטנות.

מספרים 101 קודם ל11 מפני שהמתודה לא מבצעת חישוב מי גדול יותר אלא פועלת לפי תווים 0 קודם ל1.

איך היא עובדת בפנים?

השוואה בין אלמנטים:

המתודה `sort()` משווה בין זוגות של אלמנטים סמוכים.

היא משתמשת בפונקציית השוואה פנימית (`internal comparison function`) כדי לקבוע את הסדר הנכון.

פונקציית השוואה:

הפונקציה הזו מקבלת שני ארגומנטים, שהם שני האלמנטים שמשווים.

היא צריכה להחזיר:

מספר שלילי אם האלמנט הראשון צריך להופיע לפני השני.

מספר חיובי אם האלמנט השני צריך להופיע לפני הראשון.

אפס אם אין חשיבות לסדר בין שני האלמנטים.

סידור מחדש:

על סמך תוצאות ההשוואות, המתודה מבצעת החלפות בין אלמנטים עד שהמערך כולו מסודר.

דוגמא בסיסית:

```
const numbers = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5];
numbers.sort();
console.log(numbers); // יפיק: [1, 1, 2, 3, 3, 4, 5, 5, 5, 6, 9]
```

בדוגמה הזו, לא ציינו פונקציית השוואה מותאמת, ולכן המתודה `sort()` משתמשת בפונקציה המובנית, שממירה את האלמנטים למחרוזות ומבצעת השוואה לקסיקוגרפית.

שימוש בפונקציית השוואה מותאמת:

```
const numbers = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5];
numbers.sort((a, b) => a - b);
console.log(numbers); // יפיק: [1, 1, 2, 3, 3, 3, 4, 5, 5, 5, 9]
```

במקרה הזה, פונקציית השוואה `(a, b) => a - b` מבטיחה מיון מספרי עולה מהקטן לגדול.

ובמידה ונעשה `(b-a)` איבר שני מינוס האיבר הראשון הפונקציה תחזיר מספרים בסדר יורד מהגדול לקטן.

דוגמאות נוספות לפונקציות השוואה:

מיון יורד: `(a, b) => b - a`

מיון לפי אורך מחרוזת: `(a, b) => a.length - b.length`



מיון לפי תכונה ספציפית של אובייקטים: $(a, b) \Rightarrow a.age - b.age$

נקודות חשובות לזכור:

מיון במקום: המתודה `sort()` משנה את המערך המקורי. אם ברצונך לשמור את המערך המקורי, צור עותק שלו לפני המיון.

סוג האלמנטים: המתודה `sort()` יכולה לסדר מערכים של כל סוג של נתונים, כל עוד ניתן להשוות בין האלמנטים.

יציבות: המתודה `sort()` אינה יציבה באופן כללי. זה אומר שאלמנטים שווים עשויים לשנות את מיקומם היחסי לאחר המיון.

סיכום:

המתודה `sort()` היא כלי רב עוצמה למיון מערכים ב-JavaScript. על ידי הבנת אופן פעולתה ושימוש בפונקציות השוואה מותאמות, ניתן למיין מערכים לפי קריטריונים שונים ולפתור מגוון רחב של בעיות.

Reverse

הפונקציה `reverse()` ב-JavaScript משמשת להיפוך סדר האיברים במערך. כלומר, האיבר הראשון הופך לאחרון, והאחרון הופך לראשון, וכך הלאה. הפונקציה מבצעת את השינוי ישירות על המערך המקורי, ולא יוצרת מערך חדש.

```
const numbers = [1, 2, 3, 4, 5];
numbers.reverse();
console.log(numbers); // יפיק: [5, 4, 3, 2, 1]
```

לולאות

לולאה היא פעולה שמתבצעת בצורה מחזורית עד לסיום ריצה על כל האיברים שהגדרנו בלולאה.

כללים שיש לבצע בכתיבת לולאה:

- אתחול – נגדיר ערך ראשוני איתו נתחיל לרוץ
- יעד – נגדיר ערך שבו הלולאה תפסיק לרוץ
- מקדם – נקדם בכל ריצה את הערך הראשוני עד שיגיע לערך יעד של הפסקת ריצה.

יש לציין שבתכנות ככל שהקוד יהיה קצר יותר כך תהיה לנו פחות שליטה בקוד וכלל שיהיה ארוך יותר כך תהיה לנו שליטה רבה יותר בקוד, בכל אופן יש לעבוד עם קודים קצרים ויעילים ככל הניתן יש להשתמש בשמות עם משמעות כדי שבמידה ונרצה לאחר זמן מה לגשת לקוד שכתבנו בעבר או מישהו אחר יקבל גישה לקוד שכתבנו הקוד יהיה ברור וקל להבנה.



בואו נראה כעת דוגמאות ללואות שונות שמבצעות את אותה הפעולה, לצורך הדוגמא נבנה מערך של קורסים המכיל אובייקטים עם כמה מאפיינים בתוכו:

```
let courses = [  
  {id: "01", name: "maor", age: 34},  
  {id: "02", name: "moshe", age: 80},  
  {id: "03", name: "yossi", age: 60},  
]
```

לולאת FOR

```
for (let i=0; i<courses.length; i++) {  
  console.log(courses[i]);  
}
```

לולאת FOR IN

הלולאה "המלכה" IN - רצה על כל האינדקסים במערך.

היא לבד מגדירה תחילה את | שווה לאפס ולבד מעלה את | באחד כדי לעבור כל פעם על אינדקס אחר במערך. בלולאה זו אנחנו מאבדים את היכולת לרוץ על חלק מאיברי המערך.

```
for (let i in courses) {  
  console.log(courses[i].name);  
}
```

לולאת FOR OF

הלולאה FOR OF מתייחסת לכל איבר במערך וניתן להשתמש בו בקוד.

נניח שיש לנו מערך של אובייקטים ואנחנו רוצים לגשת לאיבר בתוך המערך נכתוב זאת כך:

```
for (let course of courses) {  
  console.log (course.name);  
}
```

שמירה מקומית

זיכרון מקומי עובד בצורה של מפתחות וערכים.

ישנם 2 סוגים של זיכרון מקומי:

sessionStorage - זיכרון מקומי לטווח קצר נשמר אך ורק לאותו ששן עד לסגירת חלון הדפדפן.

localStorage - זיכרון מקומי לטווח ארוך נשמר גם לאחר סגירת הדפדפן וטעינתו מחדש.

כאשר אנו עובדים עם זיכרון מקומי הדבר הראשון שנבדוק הוא האם כבר קיימים ערכים בזיכרון המקומי.



השמה/שמירה בזיכרון לטווח ארוך localStorage:

כאשר אנחנו רוצים לבצע שמירה בזיכרון נצטרך להטמיע בתוך הסוגריים תחילה שם המפתח ולאחר מכן את הערך שישמר בתוכו.

```
localStorage.setItem(key, value);
```

קבלת ערך של מפתח בזיכרון המקומי:

כאשר אנחנו רוצים לייבא ערכים מהזיכרון המקומי נשתמש ב localStorage.getItem ובתוך הסוגריים נרשום את שם המפתח בלבד.

```
localStorage.getItem (key);
```

שמירה של אובייקט:

מכיוון שלא ניתן לשמור ישירות אובייקט בזיכרון מקומי נצטרך להמיר אותו לג'ייסון.

```
let myStorage = localStorage.setItem("user", JSON.stringify(user));
```

קבלה של אובייקט מתוך הזיכרון:

```
let myStorage = JSON.parse(localStorage.getItem("user"));
```

מחיקה מהזיכרון המקומי:

```
localStorage.removeItem("user");
```

זיכרון לטווח קצר ssessionStorage יעבוד באותה הצורה שבה הזיכרון המקומי לטווח ארוך עובד אך במקום לציין localStorage אנו נקרא ל ssessionStorage.

אם נדפיס typeof(storge) בקונסול - נקבל ערך מסוג פונקציה.

לפני שימוש בזיכרון המקומי עלינו לבדוק האם כבר קיים זיכרון מקומי למשתמש.

JSON – Java Script Object Natation

JSON היא שפה משותפת לכל שפות התכנות.

היא נכתבת כמחרוזת String ניתן להמיר אותה לקוד למשל JS והתהליך עובד גם הפוך זאת אומרת ניתן להמיר קוד JS לשפת JSON.

מכיוון שהשפה משמשת לאחסון והעברת מידע ונתונים היא משרתת אותנו רבות ב- API ומקבלת את הערכים הבאים:

.Number, String, Boolean, Literal Object, Null

ג'ייסון משפרת פלאים את מהירות התגובה בקוד היא תוספת מעט מקום בזיכרון מכיוון שמדובר במחרוזת בנויה אך ורק מ-String.



דגשים לשימוש בסינטקס של השפה:

- הנתונים מאוחסנים בצורה של שם מאפיין וערך.
- הנתונים מופרדים בפסיקים.
- סוגריים מסולסלים מאחסנים אובייקטים סוגריים מרובעים מאחסנים מערכים.
- מערך יכול להכיל אובייקטים ואובייקטים יכולים להכיל מערכים.
- נראית כמו אובייקט ליטרלי היא מתארת את עצמה וקלה להבנה.
- נתחום את המחרוזת כולה בבאק טיק.
- כל אחד מהמאפיינים וערכים שהם לא מספרים או בוליאנים יהיו בתוך גרשיים.
- סוגריים ואופרטורים למינהם אינם צריכים להיות בתוך גרשיים.

דוגמא לצורה בה נכתבת השפה ג'ייסון:

```
// דוגמא לאובייקט ליטרלי
let user = {
  id: "01",
  age: 34,
  email: "maorgross247@gmail.com",
  loggedIn: true,
  adress: {
    city: "megadim",
    street: "hahofim",
    company: "israel"
  },
  categories: ["FullStack", "Web & Mobile"]
}

// הפיכת האובייקט למחרוזת ג'ייסון
let myJson = JSON.stringify(user);
console.log(myJson);

// כעת נראה כיצד האובייקט יודפס בקונסול כג'ייסון
{ "id": "01", "age": 34, "email": "maorgross247@gmail.com", "loggedIn": true, "adress": { "city": "megadim", "street": "hahofim", "company": "israel" }, "categories": ["FullStack", "Web & Mobile"] }
```

המרות JSON

כפי שאמרנו ניתן להמיר קוד JS למחרוזת ג'ייסון וניתן להמיר JSON לקוד JS או למעשה לכל שפת תכנות קיימת מכיוון שהיא שפה משותפת לכל השפות ניתן להעביר מידע בין מערכות שעובדות בשפות שונות.

JSON.parse – המרת ג'ייסון (string) ל-JS.

JSON.stringify – המרת JS לג'ייסון (string).



שלב 2 – JS DOM

DOM

מודל האובייקטים של המסמך **Document Object Model**. ה DOM הוא למעשה ממשק API שמאפשר לתקשר עם מסמכי HTML ו-XML. ה DOM הופך את המסמך שלנו מאופן סטטי של תגים והוראות, לבנה דינאמי של אובייקטים שניתן לשנות ולשלוט בהם באמצעות JAVA SCRIPT. כאשר הדפדפן טוען מסמך HTML הוא יוצר מבנה היררכי של אובייקטים המתאים למבנה המסמך. כל אלמנט ב HTML הופך לאובייקט ב DOM עם תכונות ושיטות משלו. גישה נפוצה לגשת ל DOM היא באמצעות document זהו למעשה עץ תהליכים היררכי שבנוי בהתאמה לאלמנטים ב HTML.

לדוגמא ניתן להוסיף תוכן לתוך ה HTML בעזרת הפקודה write כמו בדוגמא הבאה:

```
document.write("hello world");
```

כברירת מחדל במידה ונוסיף תוכן נוסף הוא יתווסף לפני התוכן הקיים בדף.

String

מחרוזת בדרך כלל אנו מגדירים בתוך משתנה עם גרשיים. כאשר אנו רוצים להטמיע משתנה בתוך המחרוזת נוכל לרשום זאת ע"י חיבור מחרוזות עם הסימן + לשמות או ע"י באקטיק ושימוש בדולר בצורות הבאות:

```
let name = "maor";  
console.log("hello " + name + " nice to see you!");  
console.log(`hello ${name} nice to see you!`);
```

למחרוזת יש אורך זאת אומרת במידה ויש לנו מחרוזת לדוגמא:

```
let fullName = "maor gross"
```

אזי אורך המחרוזת יהיה שווה ל 10 תווים לבדוק זאת ע"י המילה length :

```
console.log(fullname.length);
```

דפנדנסי - בעברית ניתן לתרגם תלות, הם קודים מוכנים שניתן להלביש אל הקוד שלנו ניתן לדמיין כמו בית שבונים ומשתמשים בחלונות דלתות וצינורות אין צורך לבנות אותם אלא רק להתשמש בהם, זה למעשה להוסיף מרכיב חדש למערכת כדי לספק פונקציונליות ספציפית.

node

כל ישות במסמך ה html היא Node. בין אם זה אלמנט טקסט תכונה הערות ואפילו המסמך עצמו. כל Node יכול להכיל Nodes אחרים ויוצר את המבנה ההיררכי של ה DOM.

אלמנט



סוג מסוים של Node. מייצג תג HTML כמו DIV/P/H1 לכל אלמנט יש תכונות attribute ושיטות method משלו.

Property

תכונה של אובייקט, מגדירה מאפיין מסוים של האובייקט לדוגמה id / class / textContent. הם תכונות של אובייקטים ב-DOM.

method

פעולות שניתן לבצע על מערכים ואובייקטים הם שיטות של אלמנטים ב-DOM.

מתודה (Method) ב-JavaScript היא פונקציה הקשורה לאובייקט מסוים. כלומר, היא פעולה שניתן לבצע על אובייקט זה. לדוגמה, אם יש לנו אובייקט המייצג מכונית, המתודות שלו יכולות להיות: year, start, stop וכו'.

במילים פשוטות, מתודה היא פעולה שאתה מבצע על משהו. לדוגמה:

```
// מתודה של מסמך
document.getElementById("user");

let user = {
  firstName: "maor",
  lastName: "gross",
  id: "247",
};

// מתודה של אובייקט
user.id.length = 3 ? console.log("true") : console.log("false");

// מתודה של מערך
let arr = [];
arr.push("new index");
```

דוגמאות נוספות לרשימה נפתחת בהשמת נקודה אחר האובייקט/מעריך:

```
35 document.|
36
```

- ATTRIBUTE_NODE (property) Node.ATTRIBUTE_NODE: 2
- CDATA_SECTION_NODE
- COMMENT_NODE
- DOCUMENT_FRAGMENT_NODE
- DOCUMENT_NODE
- DOCUMENT_POSITION_CONTAINED_BY
- DOCUMENT_POSITION_CONTAINS
- DOCUMENT_POSITION_DISCONNECTED
- DOCUMENT_POSITION_FOLLOWING
- DOCUMENT_POSITION_IMPLEMENTATION_SPECIFIC
- DOCUMENT_POSITION_PRECEDING
- DOCUMENT_TYPE_NODE



```
28 let user = {
29   firstName: "maor",
30   lastName: "gross",
31   id: "247",
32 };
33
34 user.
35
```

- firstName (property) firstName: string
- id
- lastName
- abc LastName
- abc console
- abc fullName
- abc log
- abc name
- abc num1
- abc num2
- abc numbers
- abc numbers1

```
28 let arr = [];
29 arr.
30
```

- concat (method) Array<any>.concat(...items: Conc...
- copyWithin
- entries
- every
- fill
- filter
- find
- findIndex
- flat
- flatMap
- forEach
- includes

מדוע מתודות חשובות?

אירגון קוד: מתודות מאפשרות לנו לארגן קוד בצורה מודולרית, כך שכל פעולה קשורה לאובייקט מוגדרת בתוך המתודה המתאימה.

שימוש חוזר בקוד: ניתן להשתמש באותה מתודה מספר פעמים על אובייקטים שונים מאותו סוג.

הבנה טובה יותר של הקוד: מתודות הופכות את הקוד לקל יותר להבנה, מכיוון שהן מתארות פעולות ברורות על אובייקטים.



תפיסת אלמנטים

תפיסת אלמנטים באמצעות קלאסים ותגיות, ניתן להוסיף לאחר הסוגרים העגולים סוגריים מרובעים ובתוכן לציין את מיקום האלמנט כאשר 0 יהיה המיקום הראשון, 1 יהיה המיקום השני וכן הלאה בדומה למערכים.

כאשר נשתמש בTAG NAME נצטרך לתת לתגית את האינדקס שלה

```
document.getElementsByClassName("item")[0].setAttribute(id, 0);
```

שינוי מאפייני אלמנטים

לאחר שתפסנו אלמנט ניתן להגדיר לו מאפיינים חדשים, לדוגמא:

```
document.getElementsByClassName("item").id = 0;  
document.getElementsByClassName("item").src = "www.JS.com";  
document.getElementsByClassName("item").alt = "URL";
```

קבלת ערך אלמנטים – value

ניתן לקבל ערך של אלמנט לאחר תפיסתו באמצעות המילה value.

בדוגמא הבאה נראה כיצד ניתן לקבל ערך של אלמנט לתוך משתנה שנקרא username:

```
let userName = document.getElementById("userName").value;  
console.log (userName);
```

שימו לב, השמת ערך אחר במשתנה יעדכן את המשתנה ולא את הvalue של האלמנט. במידה ונרצה לאפס את המשתנה הדבר יכתב בצורה הבאה:

```
userName = "";
```

יש לציין שמשתנה שהגדרנו כעת ישרת אותנו אך ורק לקבלת הערך של האלמנט username, על מנת שנוכל לבצע שימושים נוספים באלמנט נצטרך תחילה לתפוס אותו ולאחר מכן לקבל ערך או לבצע שינויים כרצוננו וכתובה נכונה יותר תראה כמו בדוגמאות הבאות:

```
// תפיסת אלמנט  
let userName = document.getElementById("userName");  
// קבלת ערך  
console.log(userName.value);  
// איפוס ערך  
userName.value = "";  
// שינוי טקסט  
userName.textContent = "Hello";  
// עדכון HTML  
userName.innerHTML = `

# Hello World</h1>`


```



תפיסת אלמנטים באמצעות QUERY SELECTOR חובה להוסיף סימן מזהה לאלמנט בתוך הסוגריים.

ID - #

CLASS - .

```
// class
document.querySelector(".element").innerHTML = "hello world";
// id
document.querySelector("#element").innerHTML = "hello world";
```

עיצוב אלמנטים

ניתן להגדיר סטייל לאלמנט כמו בדוגמא הבאה:

```
document.getElementById('element').style.color = "green";
```

יצירת אלמנטים בDOM

באמצעות הDOM ניתן ליצור אלמנטים חדשים שיתווספו ישירות לHTML לצורך הדוגמא ניצור אלמנט DIV:

```
document.createElement("div");
```

ניתן גם ליצור אלמנט ילד להורה (אלמנט בתוך אלמנט) כמו בדוגמא הבאה:

```
document.getElementById("section").appendChild("p");
```

למעשה תפסנו אלמנט DIV שיש לו ID סקשיין ויצרנו לתוכו אלמנט מסוג פסקה.

הסרת אלמנטים בDOM

ניתן גם להסיר אלמנטים ע"י המתודה REMOVE לאלמנט או REMOVECHILD לאלמנט ילד של הורה כמו בדוגמא הבאה נניח ויש לנו אלמנט פסקה בתוך אלמנט SECTION ניתן להסיר ישירות את אלמנט P כמו בדוגמא הבאה:

```
document.querySelector("p").remove();
```

או לחלופין ניתן להשתמש במתודה appendChild על אלמנט הבאה לדוגמא:

```
document.querySelector("section").removeChild("p");
```

וכאן למעשה מחקנו את אלמנט הפסקה מתוך אלמנט ההורה.

החלפה בין אלמנטים REPLACE

ניתן להחליף אלמנטים קיימים ע"י המתודה הנ"ל זאת אומרת לתפוס אלמנט למשל מסוג כותרת h1 ולהחליף אותו לפסקה.



```
document.querySelector(".container").replaceChild(newElement,  
oldElement);
```

שימוש באירועים EVENT

ניתן להשתמש בפונקציה מובנית ONCLICK מיד לאחר המשתנה כמו בדוגמה הבאה:

```
const element = document.querySelector("#element");  
element.onclick = function () => {  
  element.style.color = "red";  
}
```

*כך למעשה שינינו צבע של אלמנט בעת לחיצה עליו.

באמצעות המתודה `addEventListener` נוכל להאזין לאירועים מגוונים.

בתוך הסוגריים של המתודה נצטרך להגדיר 2 דברים: הראשון לאיזה אירוע להאזין והשני מה יקרה בעת ביצוע הפעולה.

```
element.addEventListener(event, function);
```

נשתמש במילה `THIS` על מנת להציע על האלמנט שעליו מתרחש האירוע כמו בדוגמה הבאה:

```
element.addEventListener('click', ()=> {  
  console.log(this);  
});
```

במקרה הזה `THIS` יתן לנו את `ELEMENT`.

נוכל להשתמש במילה `THIS` כדי לגשת ישירות לאלמנט ולבצע בו פעולות נוספות כמו למשל:

```
element.addEventListener("click", function () => {  
  this.style.color = "green";  
});
```

הגדרת אירוע ושימוש בו:

```
// תפיסת אלמנט והשמה שלו בתוך משתנה  
let text = document.querySelector(".text");  
// הגדרת סוג האירוע שיתחפש על האלמנט והגדרת פרמטר אירוע כדי שנוכל להשתמש בו בתוכנית  
text.addEventListener('keyup', (event) => {  
  // שימוש בפרמטר שיצרנו  
  para.innerText = event.target.value;  
})
```

כעת נסביר על קטע הקוד שראינו כאן:

`keyup` הוא סוג של אירוע ב-JavaScript שמתרחש כאשר משתמש משחרר מקש מהמקלדת לאחר שלחץ עליו. כשאתה מוסיף מאזין לאירוע הזה באמצעות



addEventListener, אתה למעשה אומר ל-JavaScript: "כאשר מישהו משחרר מקש כלשהו, בצע את הפעולה הזו".

Para שם משתנה.

event.target: מתייחס לאלמנט HTML הספציפי שעליו התרחש האירוע. זה יכול להיות כפתור, שדה קלט, תג <div> או כל אלמנט HTML אחר.

value: תכונה זו משמשת בדרך כלל כדי לקבל את הערך הנוכחי של אלמנט. אם event.target הוא שדה קלט (כמו <input> או <textarea>), אז event.target.value יחזיר את הטקסט שהוכנס לשדה.

אובייקטים

הכרות + מאפיינים + מתודות + this

אובייקטים אנחנו מגדירים ע"י סוגריים מסולסלים. בתוך הסוגרים ניתן לאובייקט מפתח וערך. בצורה הבאה:

```
let user = {name: "maor"};
```

בדוגמא הזו:

USER הוא שם המשתנה

NAME הוא key (המפתח) או property (מאפיין)

MAOR הוא הערך שנתנו למפתח.

אובייקטים מאחסנים מידע בדומה למערכים, רק שבמערכים במידע נשמר בתוך אינסדקסים ובאובייקטים המידע נשמר במתוך מפתחות וערכים.

כאשר אנחנו רוצים לשמור מידע במקום אחד בזיכרון עם מאפיינים כמו מזהה id, כתובת מגורים, מספר טלפון, שם משתמש, סיסמא, דוא"ל וכו' אנו נעשה שימוש באובייקטים.
לדוגמא:

```
let product = {
  id: "1",
  name: "iphone 16 pro",
  price: 5000,
  category: "smart phone"
}
```

כאשר נרצה לגשת לאחד המפתחות באובייקט אנו נעשה שימוש בשם המשתנה נוסף נקודה ולאחריו יפתחו לנו כל המפתחות של אותו אובייקט שהגדרנו מראש נציין את שם המפתח ונוכל לבצע בו שינויים לדוגמא:

```
console.log(product.price);
// ידפיס 5000
```



הוספת מפתח חדש

נציין את שם האובייקט אליו אנו רוצים להוסיף מפתח חדש לאחר מכן נקודה ואז ניצור שם מפתח חדש נוסף שווה ולאחר מכן את הערך שאנו רוצים להלביש על המפתח לדוגמא:

```
product.color = "black";  
// יוסיף מפתח חדש עם צבע שחור
```

החלפת ערך עדכון של מאפיין

תתבצע בדומה להוספת מפתח חדש רק שהפעם אנחנו משתמשים במפתח קיים ולתוכו אנו יוצקים ערך חדש למשל נניח שקיים עדכון במחיר האיפון מ5000 ל5500 נרשום זאת בצורה הבאה:

```
product.price = 5500;  
// יעדכן את הערך הקיים 5000 למחיר חדש 5500
```

מחיקת מפתח (מאפיין) + ערך

בJava Script קיימת מילה שמורה בשם delete אנו נשתמש במילה זו על מנת למחוק את המאפיין והערך שאנחנו רוצים למשל:

```
delete product.category;
```

כעת האובייקט שלנו יראה כמו קודם רק ללא המפתח category

```
> delete product.category;  
< true  
> product  
< {id: '1', name: 'iphone 16 pro', price: 5500, color: 'black'}  
  color: "black"  
  id: "1"  
  name: "iphone 16 pro"  
  price: 5500  
  ► [[Prototype]]: Object  
>
```

אובייקט של מערך

```
let company = {  
  id: "12345",  
  name: "marketing solutions bussines",  
  departments: ["HR", "IT", "software", "admin"]  
}
```



בדוגמא זו אנו רואים מפתח בשם departments שמכיל מערך של ערכים.

אם נרצה להדפיס איבר מהמערך נשתמש בשם האובייקט + נקודה + שם המפתח ובסוגריים מרובעות נזין את מספר האינדקס במערך לדוגמא:

```
console.log(company.departments[0]);  
// HR ידפיס את האינדקס הראשון במערך
```

במידה ונרצה לעשות שימוש בכל האיברים במערך כמו למשל להדפיס אותם לקונסול נצטרך ליצור לולאה שתרוץ על כל אורך המערך כמו בדוגמא הבאה:

```
for (let i = 0; i < company.departments.length; i++) {  
  console.log(company.departments[i]);  
}
```

אובייקט שמכיל מערך של אובייקטים:

```
let data = {  
  company: "marketing solutions bussines",  
  users: [  
    {name: "maor", age: 34},  
    {name: "reut", age: 29},  
  ]  
}
```

בואו נראה כיצד עלינו לעבוד עם אובייקט מהסוג הזה.

Read – במידה ונרצה להדפיס לקונסול את השם של האיבר הראשון במערך users עם המפתח name נכתוב זאת כך:

```
console.log (data.users[0].name);  
// ידפיס מאור
```

Create – במידה ונרצה ליצור איבר חדש במערך של האובייקט שנקרא users נצטרך להשתמש במתודה push() כמו בדוגמא הבא:

```
data.users.push({name: "יוסי", age:45});
```

כעת הוספנו איבר חדש במערך המכיל אובייקט שיש לו מפתח name יוסי ומפתח גיל 45. כך האובייקט החדש שלנו יראה בהדפסה לקונסול:



```
> data.users.push({ name: "יוסי", age: 45
  });
< 3
> data
< {company: 'marketing solutions bussine
  s', users: Array(3)} i
  company: "marketing solutions bussine
  ▼ users: Array(3)
    ▶ 0: {name: 'מאור', age: 34}
    ▶ 1: {name: 'רעות', age: 29}
    ▶ 2: {name: 'יוסי', age: 45}
    length: 3
    ▶ [[Prototype]]: Array(0)
    ▶ [[Prototype]]: Object
>
```

מתודה בתוך אובייקט

במידה ונרצה שהאובייקט שלנו יכיל בתוכו מתודה/פונקציה ניתן להגדיר זאת במגוון צורות להלן הצורה הבסיסית:

```
let user = {
  name: "maor",
  email: "maorgross247@gmail.com",
  password: "12345",
  loggedin: false,
  getdetails: function () {
    console.log(`name: ${user.name}
email: ${user.email}`);
  },
};
```

כעת במידה ונרצה להפעיל את הפונקציה שלנו שהיא אנונימית למעשה שמורה בתוך מפתח הנקרא בשם getdetails נצטרך לכתוב זאת ע"י שימוש תחילה בשם המשתנה user + נקודה + שם המפתח בתוספת סוגריים להפעלת הפונקציה `getdetails()`.

וכך זה יראה בקונסול:



```
> user.getdetails()
name: maor                               script.js:49
email: maorgross247@gmail.com
< undefined
>
```

שימוש ב this

על מנת לאפשר העתקה של האובייקט שכולל בתוכו את הפונקציה ולדאוג לכך שהיא תיהיה גנרית (ניתנת לשימוש חוזר) נעשה שימוש במילה this שלמעשה מצביעה על אותו אובייקט שאנחנו משתמשים בו כעת ובכך ניתן יהיה לשכפל את הפונקציה ללא שינויים בשמות המפתחות למשל בדוגמא הבאה:

```
let user = {
  name: "maor",
  email: "maorgross247@gmail.com",
  password: "12345",
  loggedin: false,
  getdetails: function () {
    console.log(`name: ${this.name}
email: ${this.email}`);
  },
};
```

איך ניגש לאובייקט באמצעות מפתח ?

המתודה findIndex עוברת על כל הערכים במערך ומחזירה את האינדקס של אותו הערך לדוגמא:

```
let users = [
  {id: "1", name: "maor"},
  {id: "2", name: "yossi"},
  {id: "2", name: "moshe"},
]

let findUser = users.findIndex(obj => obj.id == 2);
```

עדכון ערך:

```
users[findUser].name === "maor";
```



Object Destructuring – פירוק אובייקט

לרכיביו

כשאנחנו מפרקים אובייקט חייבים אנחנו חייבים להשתמש בשמות זהים למאפיינים של האובייקטים. הסדר של המפתחות לא חשוב לנו. הפירוק למעשה יוצר משתנים חדשים ומכניס לתוכם עותק של הערכים שיש בתוך המפתחות.

פירוק אובייקט יתבצע עם סוגריים מסולסלים:

```
// שם האובייקט = איברים שאנחנו רוצים לפרק/  
let {id,age} = user;  
console.log (id, age);
```

array Destructuring – פירוק מערך

לרכיביו

כאן הסדר מאוד חשוב לנו מאחר ומדובר באינדקסים של המערך, השמות לא חשובים.

```
let array = [  
  1, "maor", true  
];  
  
let [id, _name, loggedIn] = array;  
console.log(id, _name, loggedIn);
```

שלב 3 – JS OOP

OOP – Object Oriented Program מבוא לתכנות מונחה עצמים

עצם – אובייקט שנוצר מתוך מחלקה.

Literal object – נוצר באופן רגיל ידני.

Instance object related – נוצר מתוך מחלקה באופן אוטומטי ודינמי בלי התערבות של המתכנת בקוד.

מחלקות ואובייקטים (By reference)

מחלקה היא תבנית ליצירת אובייקטים.

כאשר אנחנו משתמשים במידע שאין לו מכנה משותף כמו רשימת שמות מספרי טלפון כתובת אנחנו נשמור את המידע באובייקט.



כדי לשמור את המידע באופן אוטומטי נבנה מחלקות.
Js vanilla es5 2009 יוצרים את המחלקה באמצעות פונקציה.
עד היום הגדרנו אובייקט באופן ידני בצורה הבאה:

```
// literal object  
let s1 = {  
  id: "1",  
  name: "maor"  
}
```

כעת נגדיר אובייקט שנוצא מתוך מחלקה:

```
// נוצר מתוך מחלקה  
// מחלקה סטטית  
function Student() {  
  this.id = "1";  
  this.name = "maor";  
}  
  
let s1 = new Student();
```

הפעלת הפונקציה `new Student()` יוצרת אובייקט חדש.

מחלקה דינמית (גנרית)

זוהי למעשה מחלקה שאנו מגדירים כיצד יוצר האובייקט שלנו ללא שינוי באובייקט עצמו.
למחלקה דינמית נגדיר פרמטרים וכאשר נקרא לפונקציה ניתן לה ארגומנטים שיצרו לנו את האובייקט.

```
function Student (_id, _name) {  
  this.id = _id;  
  this.name = _name;  
}  
  
let st1 = new Student ("1", "maor");
```

בצורה הזו נוכל ליצור מערך של סטודנטים באופן הבא:

```
let students = [new Student("1", "maor")];
```

נקפיד שסדר הפרמטרים יהיה סדר המפתחות זאת אומרת הפרמטר הראשון יהיה המפתח הראשון, הפרמטר השני יהיה המפתח השני וכן הלאה...
כעת נראה דוגמה לפונקציה אנונימית ופונקציה עם שם:

```
// פונקציה עם שם
```



```
function Person (name,age) {
  this.name = name;
  this.age = age;
}

//פונקציה אנונימית
let Person = function (name,age) {
  this.name = name;
  this.age = age;
}
```

בואו נגדיר את המחלקה שלנו לפי הפונקציות הנ"ל וניצור פונקציה שמכניסה את האובייקט שלנו לתוך מערך:

```
let peopels = [];
function addPerson() {
  let personName = document.getElementById("name").value;
  let personAge = document.getElementById("age").value;
  peopels.push(new Person(personName, personAge));
  console.log(peopels);
}
```

מתודות במחלקות

ניתן להוסיף מתודות למחלקות הכוונה לפונקציה שתבצע פעולה מסויימת כמו למשל לקבל ערך של מפתח או להוסיף ערך למפתח כמו למשל בדוגמא הבאה:

```
function Person (userId,UserName) {
  this.id = userId;
  this.name = userName;
  this.getName = function (){
    return `Hello ${this.name}`;
  }
  this.setName = function (newName){
    return this.name = newName;
  }
}
```

*this מתייחס אל האובייקט Person כאשר נפעיל את המתודות getName אנחנו למעשה נקבל את הערך שיש באובייקט person במפתח name.
כאשר נפעיל את המתודה setName למעשה נכנס לתוך האובייקט person למפתח name את הערך שקיבלנו כארגומנט newname לפונקציה.

```
let p1 = new Person ("12345", "מאור");
```



```
console.log (p1.getName());  
// ידפיס מאור  
p1.setName("גרוס");  
// יחליף את השם מאור בשם גרוס
```

קבלת תכונות של אובייקט בצורה דינמית

כאשר נרצה להגדיר מתודה בצורה גנרית שתיגש למאפיין מסוים מחוץ לאובייקט נגדיר בתוך האובייקט את המתודה נשתמש בthis ולאחר מכן נפתח סוגרים מרובעים המאפשרים גמישות וגישה לתכונות של אובייקט.

הסבר נעוץ בשתי נקודות עיקריות:

גמישות גישה לתכונות:

סוגריים מרובעים מאפשרים גישה דינמית לתכונות: כאשר משתמשים בסוגריים מרובעים, הערך בתוך הסוגריים מתפרש כמחרוזת המייצגת את שם התכונה. זה מאפשר לנו לגשת לתכונות של אובייקט באופן דינמי, כלומר, שם התכונה יכול להיות מאוחסן במשתנה או מחושב בזמן הריצה.

גישה לתכונות לא ידועות מראש: במקרה שלנו, הפונקציות set ו-get מקבלות את שם התכונה כארגומנט. זה מאפשר לנו להגדיר או לקבל את הערך של כל תכונה באובייקט, ללא צורך בידיעה מראש של שמות כל התכונות.

הבדל בין סוגריים מרובעים לסוגריים עגולים:

סוגריים מרובעים לגישה לתכונות: כפי שהסברנו, סוגריים מרובעים משמשים לגישה לתכונות של אובייקט.

סוגריים עגולים לקריאה לפונקציות: סוגריים עגולים משמשים לקריאה לפונקציות. כאשר כותבים this.prop, זה נחשב לקריאה לתכונה שנקראת "prop" באובייקט this. אם התכונה הזו לא קיימת, תקבל שגיאה.

לסיכום:

גמישות: סוגריים מרובעים מאפשרים לנו לכתוב קוד גמיש יותר, שמסוגל להתמודד עם מצבים שבהם שמות התכונות אינם ידועים מראש.

דינמיות: ניתן להשתמש במשתנים או בביטויים כדי לקבוע את שם התכונה בזמן הריצה.

קריאה לפונקציות: סוגריים עגולים משמשים לקריאה לפונקציות, ולא לגישה לתכונות.

```
function Product(id, name, price, category, description) {  
  this.id = id;  
  this.name = name;  
  this.price = price;  
  this.category = category;  
  this.description = description;  
  this.set = function (prop, newValue) {  
    this[prop] = newValue;  
  };  
}
```



```
// שימו לב לסינטקס סוגריים מרובעים מאפשרים לנו לגשת לתכונות של אובייקט  
}  
this.get = function (prop) {  
  return this[prop];  
}  
}
```

שימוש במתודה get מחוץ לאובייקט שיצרנו:

```
let products = [  
  new Product("2536", "Sony Pro 32GB", 500, "Storage", "The new XQD  
cards achieve Max read"),  
  new Product("2537", "Lenovo Legion 15.6 Gaming", 4228, "Laptop",  
"Lenovo Legion 5 pairs unparalleled flexibility with incredible power,  
offering a plethora of performance options for any gamer in a clean and  
minimalist design"),  
]  
//for id = Without get  
for (let i = 0; i < products.length; i++) {  
  document.getElementById("tableContent").innerHTML += `|  
    <td>${products[i].id}</td> //הדרך שבה היינו רגילים לקבל ערך עד היום/  
    <td>${products[i].get("name")}</td> // הדרך החדשה עם המתודה  
    גט שיצרנו  
  
    <td>${products[i].get("price")}</td>  
    <td>${products[i].get("category")}</td>  
    <td>${products[i].get("description")}</td>  
  </tr>`  
}

```

שימוש במתודה set מחוץ לאובייקט שיצרנו:

```
products[i].set("name", "maor");  
//כניס למפתח שם את השם מאור
```

מכיוון בנינו את האובייקט בצורה דינמית ניתן יהיה להגדיר set לכל מפתח באובייקט אנו נצטרך לציין תחילה את שם המפתח ולאחר מכן את הערך החדש.

ירושת מחלקות

בירושת מחלקות אנו יורשים למחלקת הבן את כל המאפיינים והמתודות שיש למחלקת האב.

למחלקת הבן ניתן להוסיף מאפיינים ומתודות נוספות שאין למחלקת האב.

בדרך כלל נשתמש באותם פרמטרים של מחלקת האב, ניתן להוסיף או להפחית פרמטרים.

Call קריאה למחלקת האב.

הקריאה תעשה מתוך מחלקת הבן.



הפונקציה call מקבלת:

כערך ראשון this למעשה האובייקט שיחליף את מחלקת האב

פרמטרים נוספים שיעברו לפונקציה, ושוב בכל מקרה בירושת מחלקות אנו נקבל את כל המאפיינים והמתודות של מחלקת האב, יהיה ניתן להוסיף או לדרוס מאפיינים ומתודות.

במידה ולא הגדרנו כפרמטרים חלק מהתכונות שעוברות בירושה ממחלקת האב אנו נקבל כערך באותם המפתחות undefined (לא הוגדר ערך).

אנחנו יכולים להעביר כפרמטרים מאפיינים בלבד ולא מתודות שהם למעשה פונקציות.

מחלקת הבן תירש בכל מקרה גם את המתודות.

```
function Person(id, name, age) {
  this.id = id; // this = Person
  this.name = name;
  this.age = age;
  this.sleep = function () {
    console.log(`Good night ${this.name}`);
  };
}

// מחלקת הבן
function Employee(id, name, age, department) {
  // לקרוא לבנאי של Person
  // עם הנתונים שנשלחו ל Employee
  Person.call(this, id, name, age); // this = Employee
  this.department = department;
  this.work = function () {
    console.log(`Go back to work in ${this.department}`);
  }
}

let p1 = new Person(1, "Yoav", 41);
console.log(p1);
p1.sleep();

let p2 = new Person(3, "Shira", 22);
console.log(p2);
p2.sleep()

let e1 = new Employee(2, "Sigal", 32, "HR");
console.log(e1);
e1.work();

// p1.work() // ERROR
```




כפי שהסברנו קודם ניתן לדרוס מתודה קיימת ע"י שימוש חוזר באותה המתודה עם קבלת ערכים שונים כמו בדוגמא הבאה:

```
//חלקת האב
function Animal(name) {
  this.name = name;
  this.sound = function () {
    console.log("General Sound");
  }
}

//חלקת הבן
function Cat(name, food) {
  //קריאה למחלקת האב והורשת המאפיינים למחלקת הבן
  Animal.call(this, name);
  this.food = food;
  //overriding דריסה של מתודה
  this.sound = function () {
    console.log("Meow meow");
  }
}
```

מחלקות es6 – class

- נשתמש במילה class כדי לפתוח מחלקה חדשה
- שם מחלקה נגדיר באמצעות אות גדולה
- Constructor הוא הבנאי של המחלקה למעשה פונקציה שמקבלת פרמטרים, יצירת מופעים תתבצע באמצעותו.
- This משמש ליישום הבנאי
- מתודות יהיו חלק מהמחלקה ולא בתוך ה-constructor.

```
class Person {
  constructor(id, name) {
    this.id = id;
    this.name = name;
  }
  getDetails() {
    console.log(`id: ${this.id}, name: ${this.name}`)
  }
  // קבלה והשמה של ערכים כלליים
  get(prop) {
    return this[prop];
  }
}
```



```
}  
set(prop, newValue) {  
  return this[prop] = newValue;  
}  
}
```

מתודה סטטית

מתודה סטטית נראית כמו אובייקט שנוצר באמצעות מחלקה בתוספת המילה static, לאחר שהגדרנו אותה אנחנו יכולים לקרוא לה בלי ליצור מופע. למשל כמו שיש לנו את המחלקה Math ובתוכה מתודות כמו PI, pow, min, max כל אלו למעשה הן מתודות סטטיות.

כאמור מתודה סטטית היא לא חלק ממופע של אובייקט.

לא כל מחלקה נועדה כדי ליצור אובייקטים לצורך המחשה נראה את הדוגמאות הבאות.

יצירה/הגדרה של מתודה סטטית:

בדוגמא זו circle זו המחלקה

calcArea זו המתודה

- ניתן ליצור מספר מתודות למחלקה באמצעות המילה static.
- מתודות כלליות/ספציפיות יוצרים מחוץ לקונסטרוקטור כך שיהיה חלק מהמחלקה.
- ירושה של מתודות מתבצעת באופן אוטומטי.

```
//חישוב שטח מעגל  
class Circle {  
  static calcArea(radius) {  
    return Math.PI * radius * 2;  
  }  
}  
//מקבלת רדיוס 5 ומחשבת את שטח המעגל באמצעות מתודה שיצרנו  
console.log(Circle.calcArea(5));
```

במידה והיינו רוצים ליצור את המתודה Math.pow זה היה נראה כך:

```
//מקבלת 2 מספרים ומחזירה את חזקתם  
class Math () {  
  static pow (a,b) {  
    return a**b;  
  }  
}
```



Privet – פרטי

מאפיינים מסוג פרייבט אנחנו מצהירים עליהם ברמת המחלקה.
בדוגמא שלנו password יהיה פרטי ורק אדמין יוכל לגשת למתודה שמקבלת את הסיסמא
כל משתמש אחר יקבל undefined.

```
class User {
  #password;
  constructor(id, name, password, premission) {
    this.id = id;
    this.name = name;
    this.#password = password;
    this.premission = premission;
  };
  getPassword() {
    if (this.premission === "admin") {
      return this.#password;
    }
  }
}

let u1 = new User("1", "maor", "12345", "admin");
let u2 = new User("1", "maor", "12345", "user");
//דפיס את הסיסמא
console.log(u1.getPassword());
//דפיס undefined
console.log(u2.getPassword());
```

הורשה – es6 inheritance

- נגדיר מחלקה חדשה son שיוורשת מאפייני ממחלקת father באמצעות המילה extends
- לתוך הקונסטרקטור נכניס פרמטרים שיהיו המאפיינים שנרצה לרשת ממחלקת father בתוספת של מאפיינים חדשים שיהיו למחלקת sun במידה.
- ירושה של מתודות מתבצעת באופן אוטומטי.
- לפני שאנחנו פונים להגדרת המחלקה הנוכחית (המחלקה היורשת) עלינו לפנות להגדרת המחלקה שממנה אנחנו יורשים באמצעות המילה SUPER ובתוכה המאפיינים אותם נרצה לרשת.
- המילה super חייבת להיות בשורה הראשונה לאחר הקונטרקטור (אחרת תהיה שגיאה בתוכנית), היא יוצרת את ההורשה וניתן לה את הפרמטרים שעוברים בהורשה. היא מפעילה את הקונסטרקטור של מחלקת האב.

```
class Father {
  constructor(name, age) {
    this.name = name;
  }
}
```



```
    this.age = age;
  }
}

class Sun extends Father {
  constructor(name, age, game) {
    super(name, age);
    this.game = game;
  }
}

let sun1 = new Sun("maor", 34, "CS");
console.log(sun1);
```

מבוא לתכנות א-סינכרוני

פעולות סינכרוניות הם פעולות שמתבצעות לפי סדר שורה אחרי שורה רק כאשר הפעולה תתבצע ניתן יהיה להמשיך לשורה הבאה הן תוקעות את התוכנית והתוצאה שלהם זמינה מיד, לעומת פעולות אסינכרוניות כמו קריאה לשרת הן מתבצעות במקביל לפעולות אחרות או מתעכבות בצורה שלא חוסמת את התוכנית והתוצאה שלהן זמינה בסיום הפעולה באמצעות פונקציות call back, promiss, async/await, setTimeout, מסתיימת היא תקרא לפונקציה אחרת כדי לטפל בתוצאה.

מחלקת Promise - הבטחה

Promise – מחלקה מובנת ב-javascript והיא מאפשרת לנו להבטיח שתהליך ראשון יסתיים לפני הפעלת תהליך שני.

היא מקבלת פונקציה עם 2 פרמטרים:

1. **Resolve** – כאשר יש הצלחה (כתיבה מקוצרת מקובלת **res**)
2. **Reject** – כאשר יש כשלון / שגיאה. (כתיבה מקוצרת מקובלת **rej**)

שמות אלה הם שפה מקובלת לשימוש בשמות משתנים במחלקת promise.

catch-then שאנחנו נראה פה בהמשך גם הם מקבלות פונקציה שניתן להכניס לתוכה פרמטר.

כעת נדמה פניה לשרת באמצעות setTimeout:

```
let p = new Promise((resolve, reject) => {
  //A
  setTimeout(() => {
    console.log("Login Success");
    resolve();
  }, 3000)
});
```



```
p.then(() => {  
  //B  
  console.log("Redirect...");  
});
```

resolve() מפעיל את את .then()

then() יפעל במידה ו-P הצליח.

Resolve יגיע בסוף הסקופ (קטע הקוד) ניתן לבצע פעולות בתוך הסוגריים שלו.

כעת נראה דוגמא לפעולת הצלחה בתנאי שמתקיים או פעולת כשלון בתנאי שלא מתקיים:

```
// Example  
let p2 = new Promise((res, rej) => {  
  let num = 1;  
  // res & rej get parameters  
  if (num > 2) res("Bigger then 2");  
  else rej("Lower or equal to 2");  
});  
  
p2.then((sucess) => {  
  console.log(sucess);  
}).catch((error) => {  
  console.log(error);  
});
```

בתנאי ש-num גדול מ-2 יודפס בקונסול:

Bigger then 2

בתנאי ש-num קטן מ-2 יודפס בקונסול:

Lower or equal to 2

בואו נראה דוגמא לפעולה שתבצע במידה ויותר מתהליך אחד הצליח, לשם כך נשתמש ב-
Promise.All אשר מקבלת מערך של הצלחות:

```
// Promise all  
let a = new Promise((resolve, reject) => {  
  setTimeout(() => {  
    console.log("A process begin");  
    resolve("A process done");  
  }, 2000);  
});  
  
let b = new Promise((resolve, reject) => {  
  setTimeout(() => {  
    console.log("B process begin");  
    resolve("B process done");  
  }, 3000);  
});
```



```
});  
  
Promise.all([a, b]).then((dataArr) => {  
  console.log(dataArr);  
});
```

Promise הוא אובייקט שמייצג את התוצאה הסופית (המוצלחת או הכושלת) של פעולה אסינכרונית. פעולה אסינכרונית היא פעולה שעלולה לקחת זמן מסוים להסתיים, כמו למשל בקשה לשרת, קריאה לקובץ או השהיה בזמן.

למה אנחנו צריכים Promises?

ניהול קוד אסינכרוני: Promises מאפשרות לנו לכתוב קוד אסינכרוני בצורה מסודרת יותר, ומאפשרות לנו לטפל בהצלחה או בכשלון של פעולות אסינכרוניות. שרשרת פעולות: Promises מאפשרות לנו לשרשר מספר פעולות אסינכרוניות זו אחר זו, כך שהפעולה הבאה תתחיל רק לאחר שהפעולה הקודמת הסתיימה בהצלחה.

איך עובד Promise?

Promise יכול להיות במצב אחד משלושה:

pending: הפעולה האסינכרונית עדיין מתבצעת.

fulfilled: הפעולה הסתיימה בהצלחה.

rejected: הפעולה נכשלה.

ראשית המילה Promise כותבים עם אות ראשונה גדולה, כאשר יוצרים Promise, מעבירים אליו פונקציה עם שני פרמטרים: resolve ו-reject. הפונקציה הזו תקרא ל-**resolve** אם הפעולה הסתיימה בהצלחה, או ל-**reject** אם היא נכשלה.

2 הפרמטרים האלו מתנהגים ממש כמו **return** בפונקציה שאנחנו רגילים:

הפרמטר הראשון **resolve** יחזיר הצלחה.

הפרמטר השני **reject** יחזיר שגיאה בקונסול.

then: משמש לטיפול בתוצאה המוצלחת של ה-Promise.

catch: משמש לטיפול בשגיאה שהתרחשה במהלך הפעולה.

THEN ו-CATCH תמיד מקבלים פונקציה שמקבלת פרמטר. לפרמטר של THEN ניתן שם לפי השימוש שלנו ב-PROMISE ובדרך כלל לפרמטר של CATCH נקרא בשם error מלשון שגיאה.

את הפונקציה PROMISE נשים בתוך משתנה על מנת שנוכל לקרוא לה בתוכנית ע"י שימוש בשם המשתנה, מכיוון שיכולים להיות כמה THEN נהוג לרשום אותם בשורה נפרדת מהפונקציה. כעת נראה דוגמה לפונקציית PROMISE בסיסית ופרמיטיבית:



```
const initPromise = () => {
  cleanRoom
    .then((data) => {
      alert(data);
    })
    .catch((error) => {
      alert(error);
    });
};

const cleanRoom = new Promise((resolve, reject) => {
  let isClean = true;
  if (isClean === true) {
    resolve("good job, go to the beach");
  } else {
    reject("go clean the all house!");
  }
});

initPromise();
```

דוגמא נוספת לפונקציית PROMISE מתוך הסבר של בארד AI :

```
const myPromise = new Promise((resolve, reject) => {
  // קוד שמבצע את הפעולה האסינכרונית
  setTimeout(() => {
    // אם הפעולה הצליחה
    resolve("הפעולה הסתיימה בהצלחה");
  }, 2000);
});

myPromise
  .then((result) => {
    console.log(result); // יציג "הפעולה הסתיימה בהצלחה" לאחר 2 שניות
  })
  .catch((error) => {
    console.error(error); // יופעל אם הפעולה נכשלה
  });
```

כעת נראה כיצד באמת נהוג לכתוב את הפונקציה PROMISE, מצורף הסבר בהערות:

```
// CALL BACK פונקציה
// שקוראת לפונקציה
// Promise
const initGaeden = ()=> {
```



```
// מקבלת פרמטר נכון או לא נכון
cleanGarden(true)
  .then (data => {
    alert(data);
  })
  .catch (error => {
    alert(error);
  })
}

// הגדרת הפונקציה PROMISE
// מקבלת כל פרמטר לבדיקה זה יכול להיות אובייקט מערך ועוד
const cleanGarden = (isGarden) => {
  // מחזירה הבטחה חדשה שיש לה פרמטרים של הצלחה וכישלון
  return new Promise ((resolve, reject) => {
    if (isGarden) {
      resolve("good job, play XBOX");
    } else {
      reject("not good, go to clean all street");
    }
  })
}

initGaeden();
```

דוגמא נוספת לשימוש ב-PROMISE:

```
fetch('https://api.example.com/data')
  .then(response => response.json())
  .then(data => console.log(data))
  .catch(error => console.error(error));
```

Promise יכול לשמש אותנו למשל לממשק התחברות של משתמש למערכת (עבודה עם שרת) תחילה נבדוק אם קיים יוזר במאגר הנתונים בשרת במידה וכן נוכל למשל לבדוק את הסיסמא, במידה ובקשה לא התקבלה נבצע פעולות אחרות כמו להציג את השגיאה.

לצורך המחשה הבקשה לשרת תהיה `new Promise`:

במידה וקיים יוזר נשתמש ב- `resolve`.

במידה ולא קיים יוזר נשתמש ב- `reject`.

זמן מצב תקין של קבלת נתונים מהשרת – 0.25 שניות.

Promise מאפשרת לנו לשלוט בתהליך של רצף תהליכים שלוקחים זמן לביצוע, הם ירצו ברקע עד לסיום התהליך ויבצעו פעולות שהגדרנו מראש בהתאם לקבלת הצלחה או כשלון של הבקשה.

רצף של הרבה פרומיסים נקרא `promise hell` הבא נלמד כיצד לעבוד בצורה מסודרת עם רצף של בקשות בעזרת שימוש ב- `await` ו- `async`.



סיכום:

Promises הן כלי חיוני ב-JavaScript לניהול קוד אסינכרוני. הן מספקות דרך ברורה ויעילה לטיפול בהצלחה או בכשלון של פעולות אסינכרוניות, ומאפשרות לנו לשרשר פעולות בצורה אלגנטית.

הערה:

בעוד שאין פקודה ספציפית בשם "Promise", המושג Promise מתייחס לאובייקט ב-JavaScript המשמש לניהול פעולות אסינכרוניות. הפונקציות `then`, `catch` ו-`new Promise` הן חלק מהמנגנון של Promises.

תרגום:

Promise: הבטחה / מחלקה לבקשה חדשה

async: אסינכרוני / הגדרת פונקציה אסינכרונית שתבצע פעולות כהתהליך הבקשה יסתיים

await: להמתין / לחכות עד שהבקשה תסתיים

resolve: לפתור / הגדרת פעולות כאשר בקשה הצליחה

reject: לדחות / הגדרת פעולות כאשר בקשה נכשלה.

then: אז / ביצוע פעולות כאשר הבקשה הצליחה

catch: לתפוס / ביצוע פעולות כאשר הבקשה נכשלה (לתפוס את השגיאה)

מחלקת Promise

שימוש ב-`async+await,try+catch`

- שיטה חדשה יותר לטיפול במחלקות מסוג promise.
- מחייבת אותנו להשתמש ולהגדיר פונקציות
- שיטה נוחה יותר לטיפול ברצף של מספר בקשות בזה אחר זה שממתינים לביצוע
- חובה להשתמש ב-`try` ו-`catch`, יש לציין שמילים שמורות אלו הן כלליות ולא קשורות בהכרח רק לטיפול במחלקת promise.
- `async` – הפונקציה תעבוד בתלות ל-`promise`
- `await` – המתנה עד לסיום ביצוע התהליך
- `Try` – יודעת לגשת לקבל ערך מ-`resolve` באופן אוטומטי ומתארת מה יקרא במצב של `resolve`
- `Catch` – מתאר מה יקרא המצב של `reject`, נצטרך להשתמש בפרמטר לקבלת ערך של פעולה שלא הצליחה.
- ניתן להוסיף בתוך `try` פונקציות נוספות עם המילה `await` וכך כאשר הפונקציה הראשונה תסתיים בהצלחה היא תעבור לפונקציה הבאה. כל עוד פונקציה ראשונה לא הסתיימה, פונקציה שניה לא תתחיל וכן הלאה...



- במידה ולא נוסף את המילה await או נקבל מיידית אובייקט מסוג promise כאשר נוסף את המילה await על מנת להמתין עד שהבקשה תסתיים, יבוצעו פעולות במקרה של הצלחה או כשלון כפי שהגדרנו.
- Trycatch – קיצור לבנית מוכנה בתוכנה vsCode
- השימוש שלנו בפעולות אסינכרוניות יהיה כאשר תהליך אחד תלוי בתהליך אחר שיסתיים.
- יש להגדיר resolve ו-reject מדויקים לכל תהליך כדאי לפרט על כל תהליך בצורה שנדע איזה תהליכים הצליחו ואיזה תהליך נכשל.
- New Error מחלקה שמורה בתוכנית שמציגה לנו שגיאה בקונסול וגם יודעת להגיד היכן הייתה השגיאה.

כעת נראה דוגמה לשימוש בפרומיס עם המילים השמורות שלמדנו ונפרט על כך תהליך:

```
function func(num) {
  // פונקציה שמחזירה פרומיס
  return new Promise((resolve, reject) => {
    // הגדרת פעולות במקרה של הצלחה או כשלון בהתאם לתנאי מסויים
    num > 10 ? resolve("true") : reject("false");
  })
}

// הפונקציה תעבוד בתלות לפרומיס
async function checkNum() {
  מתאר מה יקרא במצב של בקשה שהצליחה/
  try {
    // להמתין עד שהתהליך של הבקשה יסתיים
    let result1 = await func(12);
    console.log(result1);
    // קריאה נוספת לפונקציה שתתחיל מיד אחרי שהראשונה תסתיים בהצלחה
    let result2 = await func(8);
    console.log(result2);
    // מתאר מה יקרא במצב של בקשה שנכשלה
  } catch (error) {
    // מחלקה חדשה לטיפול בשגיאה
    console.log(new error(error));
  }
}
```

רשימת נושאים נוספים שיעלו בקרוב:

- Fetch
- Modules
- קריאות XMLHttpRequest
- תנאים



- עבודה עם תאריכים, טקסטים ומספרים
- `setTimeout`
- `setInterval`
- application Programming interface – API מאגר נתונים `.data`. חלק גדול מ-API יהיה במבנה של JSON.