מיקוד מבחן ב JavaScript

* ***advanced js book*** (only the part from the first exam)
* ***events*** - onload, onclick, onchange, etc...
* ***Async js*** - event loop, callback, promise, await + the 3 cases that we use async in js.
* ***http request - ajax***:
  + fetch
  + XMLHttpRequest

סיכום חומר Advanced JavaScript

**טיפוס משתנה דינמי:**

JavaScript היא שפת תכנות דינמית, ולכן אין צורך להכריז על סוג של משתנה בזמן ההצהרה. סוג המשתנה יקבע באופן דינמי בזמן ביצוע התוכנית (בזמן ריצה). כלומר, **JavaScript** **מאפשרת לאותו משתנה לקבל ערכים מטיפוסים שונים**.

**סוגי טיפוסים:**

**ניתן לחלק לשתי קטגוריות את סוגי הטיפוסים ב-JavaScript:**

1. ***טיפוסים פרימיטיביים (Primitive Types):***
   * Strings
   * Boolean
   * Number
   * Null
   * Undefine
   * Symbol (קיים בes6)
2. ***טיפוסי אובייקט (Object types):***
   * **כל טיפוס שלא נכלל בטיפוסים הפרימיטיביים, נכלל תחת object types.**
   * אובייקטים מובנים של JavaScript הנפוצים בשימוש, הם המחלקה Date המגדירה אובייקטים המייצגים תאריכים. המחלקה RegExp מגדירה אובייקטים המייצגים ביטויים רגולריים (pattern-matching). והמחלקה Error המגדירה אובייקטים המייצגים שגיאות זמן ריצה שעלולים להתרחש בתוכנת JavaScript.
   * אובייקט הוא אוסף של מאפיינים שבהם לכל מאפיין יש שם וערך (או ערך פרימיטיבי כגון מספר, או אובייקט).
   * אובייקט JavaScript רגיל, הוא unordered collection (אוסף לא מסודר) של מפתחות וערכים.
   * JavaScript מגדירה גם סוג ordered collection (אוסף מסודר), הידוע כמערך (Array) המייצג אוסף מסודר של ערכים ממוספרים ע"י אינדקס מספרי ולא ע"י מפתח מחרוזתי.
   * **פונקציה מוגדרת ב- JavaScript כטיפוס של אובייקט.**

**הערה**:

ה- interpreterשל JavaScript מבצע automatic garbage collection (מכלול אשפה אוטומטי) למטרת ניהול הזיכרון. משמעות הדבר היא כי תוכנית יכולה ליצור אובייקטים לפי הצורך, והמתכנת אף פעם לא צריך לדאוג לבצע deallocation (ניהול זיכרון דינמי) עבור האובייקטים שנוצרו.

**כאשר אובייקט אינו נגיש יותר** (לתוכנית אין עוד דרך לגשת אליו) **ה- interpreter** **יודע** **שלא ניתן להשתמש באותו אובייקט שוב**, **ומשחרר באופן אוטומטי את הזיכרון שהוקצה עבור האובייקט.**

mutable and immutable types (טיפוסים משתנים וטיפוסים בלתי ניתנים לשינוי):

* **mutable** - ערך של משתנה מסוג mutable יכול להשתנות

קבוצת האובייקטים הם mutable.

* **immutable** - number, Boolean, null, undefined string וכו', אינם ניתנים לשינוי.

ניתן לגשת לטקסט בכל אינדקס של מחרוזת, אך JavaScript אינה מספקת אפשרות לשנות את הטקסט של מחרוזת קיימת.

**Ref type VS Value type:**

* המשתנים מסוג **value type** הם משתנים פרימיטיביים (כגון: number, Boolean, string), וכאשר המשתנה נוצר בזיכרון, הוא **יכיל בתוך שטח המשתנה עצמו את הערך המושם לתוכו**.
* המשתנים מסוג **reference type** הם משתנים מטיפוס object type, והמשתנה **מכיל הפניה לשטח שבו האובייקט נוצר**.

**לשם מה חשוב לנו להבין את הנושא של value type ו-reference type ?**

1. בביצוע השוואה בין שני משתנים (ע"י == ):

* במשתני value type – ייבדק האם שני המשתנים מכילים אותו ערך.
* במשתני reference type – ייבדק האם שני המשתנים מכילים הפניה לאותו אובייקט.

1. בביצוע השמה של משתנה אחד לתוך משתנה אחר:

* במשתנה מסוג value type – **תבוצע העתקת הנתון** שבתוך המשתנה
* במשתנה מסוג reference type – **תבוצע העתקת ההפניה** אליה מצביע המשתנה.

הערה:

* כאשר נבצע שינוי על העותק של משתנה פרימיטיבי, משתנה המקור לא יושפע.
* ואילו נבצע שינוי על העותק של משתנה לא פרימיטיבי, כלומר משתנה מטיפוס אובייקט, משתנה המקור יושפעו כיוון שהוא מצביע לאותו אובייקט שתוכנו שהעותק מצביע.

**דגש**: בשליחת פרמטרים בפונקציה, מתבצעת העתקה של הפרמטר שנשלח בקריאה לפונקציה, אל הפרמטר שמתקבל על ידי הפונקציה, ןלכן ישנם שני סוגי העברות פרמטרים:

* 1. **העברת פרמטר by value-** תתבצע כאשר נעביר משתנים פרימיטיביים.

קוד שקורא לפונקציה ושולח לה ערך של תוכן משתנה שתוכנו מועתק לתוך הפרמטר הלוקלי של הפונקציה.

* 1. **העברת פרמטר by reference**- תתבצע כאשר נעביר לפונקציה משתנה שהוא לא פרימיטיבי.

הקוד שקורא לפונקציה שולח לה ערך של כתובת למשתנה מסוים בזיכרון, והכתובת מועתקת לתוך הפרמטר הלוקלי של הפונקציה.

**== vs operator === operator:**

* **Abstract Comparison** – מיוצג ע"י אופרטור == המשמש לבדיקת השוויון בין שני ערכים לפי תוכנם.
* **Strict Comparison** – מיוצג ע"י אופרטור === המשמש לבדיקת השוויון בין שני ערכים לפי תוכנם ולפי סוג הטיפוס שלהם.

**אופרטור Typeof:**

* אופרטור typeof נועד כדי למצוא את סוג משתנה.

לדוגמה:

//typeof string

console.log(typeof "John bryce");

//typeof number

console.log(typeof 3);

console.log(typeof 3.5);

console.log(typeof Infinity);

console.log(typeof NaN);

//typeof boolean

console.log(typeof true);

console.log(typeof false);

//typeof object

console.log(typeof [1, 2, 3, 4]);

console.log(typeof { name: 'John', age: 34 });

console.log(typeof /^[0-9]$/);

console.log(typeof (new Date()));

console.log(typeof null); // returns object and this is bug in ECMA script5

//typeof undefined

console.log(typeof undefined);

//typeof function

console.log(typeof function () { });

**ההבדל בין === ל- typeof:**

לעיתים קרובות נרצה לבדוק האם משתנה מסוים מכיל את הערך undefined.

ניתן לבצע את הבדיקה במספר דרכים:

* **בדיקה באמצעות ===:**

**var x;**

if (x === undefined) {

document.write("x is undefined");

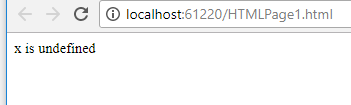
}

else {

document.write("x is defined")

}

כאשר נריץ את הקוד בדפדפן, נקבל את התוצאה הבאה:



* **בדיקה באמצעות typeof:**

if (typeof x === 'undefined') {

document.write("x is undefined");

}

else {

document.write("x is defined")

}

נקבל שגיאה של הפניה שגויה, כי המשתנה לא הוגדר מהו לפני הרצה.

if (x === undefined) {

document.write("x is undefined");

}

else {

document.write("x is defined")

}

**סיכום:**

* typeof יכול גם לעבוד על משתנה undefined
* === יזרוק ReferenceError עבור המשתנה undefined

**טיפוס number:**

* JavaScript אינה עושה הבחנה בין ערכים שלמים וערכים ממשיים.

**מספר מילולי (number literals):**

* **מספרים שלמים**

לדוגמה: 123

* **ערכים הקסדצימליים (בסיס 16):**

hexadecimal literal מתחיל **עם0x** או **0X** ואחריו מספר הקסדצימלי.

ספרה הקסדצימלית מייצגת ערכים בטווח 0 עד 15, ויכולה להיות מיוצגות באמצאות אחת מהערכים הבאים:

* + 0-9
  + A-F
  + a-f

לדוגמה: 0xff

* **Floating-point literals** – יכולים להיות מספר המכיל decimal point(נקודה עשרונית)

לדוגמה: 3.14

**טיפוס string:**

* מחרוזת היא רצף מסודר של תווים, כאשר כל תו מורכב מ 16 סיביות.

אורך מחרוזת הוא מספר הערכים של 16 סיביות שהוא מכיל.

* ל- JavaScript אין סוג מיוחד המייצג תו אחד של מחרוזת. וכדי לייצג ערך תו יחיד יש להשתמש במחרוזת בעלת אורך של תו אחד.
* **ב- ECMAScript 5** -ניתן לטפל במחרוזות כמו read-only arrays, **ויש אפשרות לגשת לתווים בודדים של מחרוזת באמצעות סוגריים מרובעים במקום בשיטת charAt ().**

לדוגמה:

var s = "hello, world";

ניתן לגשת ככה:

console.log(s[0]);

במקום לגשת ככה:

console.log(s.charAt(0));

תבנית מחרוזות (JavaScript Templet String Literals):

* בכל פרויקט יהיו שימושים רבים באינטרפולציה(הגדלת דבר מה על ידי הכנסת חומר חדש) על מנת לשתול ערכים לתוך מחרוזת.

הדרך הסטנדרטית לעשות זאת ב-JavaScript היא באמצעות – פעולות חוזרות ונשנות(repeated concatenations).

לדוגמה:

var firstName="Anna";

var lastName = "Karp";

var str = 'firstName: ' + firstName + ', lastName: ' + lastName;

console.log(str);

* ECMAScript 2015 מביא פתרון הרבה יותר אינטואיטיבי וקל לשימוש:

var firstName="Anna";

var lastName = "Karp";

var str = `firstName: ${firstName}, lastName: ${lastName}`;

console.log(str);

תכונה נוספת של תחביר זה, היא תמיכה ב- multiline:

var firstName="Anna";

var lastName = "Karp";

var str = `

firstName: ${firstName},

lastName: ${lastName}`;

console.log(str);

**: Booleanטיפוס**

**ערך בוליאני מייצג True או false**.

כל ערך JavaScript ניתן לייצוג על ידי ערך בוליאני.

הערכים הבאים מייצגים את הערך false:

* undefined
* null
* +0
* -0
* NaN
* ""
* false

**כל שאר המספרים, האובייקטים (ומערכים) מייצגים את הערך true.**

**טיפוס null ו-undefine:**

**null** הוא language keyword המייצג ערך מיוחד המשמש בדרך כלל לציון העדר ערך.

* הפעלת האופרטור typeof על null מחזירה את המחרוזת object

**undefine** הוא הערך של כל משתנה שלא אותחל והוא מייצג גם את הערך המוחזר מפונקציות שאין להן ערך מוחזר.

Undefined הוא משתנה גלובלי מוגדר מראש (לא language keyword כמו null).

* הפעלת האופרטור typeof על undefined מחזירה את המחרוזת undefined

null ו- undefined שניהם מסמלים על היעדר ערך ויכולים לשמש לעתים קרובות תחליף אחד לשני. אופרטור השוויון == מחשיב אותם שווים. (ואילו האופרטור === מחשיב אותם כשונים). אולם נפוץ להשתמש ב- undefined כדי לייצג היעדר ערך ברמת המערכת. וב- nullלמטרת איפוס אובייקטים שכבר אותחלו.

**: Wrapper Objects**

בכל פעם שמנסים לגשת לproperty- (ערך) של מחרוזת, JavaScript ממירה את ערך המחרוזת ל- object(כמו האובייקט שנקבל על ידי הפונקציה new String() ).

האובייקט הזה יורשmethods (פונקציות) שלstring ומשמש בתור property reference (הפניה של ערך). לאחר שהשימוש בproperty- או ב-method הסתיים, האובייקט החדש שנוצר נמחק בצורה אוטומטית.

המספרים והבוליאניים משתמשים באותה שיטה: אובייקט זמני נוצר באמצעות הבנאי Number () או Boolean ()ובאמצעות אובייקט זמני זה אפשר לגשת ל property או ל- method הרצויים.

האובייקטים הזמניים שנוצרו בעת גישה למאפיין של מחרוזת, מספר או בוליאני ידועים כ- wrapper objects, ומאפייניהם הם לקריאה בלבד. ולכן אם ננסה להגדיר את הערך של property, הניסיון הזה לא יבוצע (silently ignored) מפני שהשינוי נעשה על האובייקט הזמני.

אין wrapper objects עבור ערכי null ו undefined-, ולכן כל ניסיון לגשת למאפיין של אחד מערכים אלה יגרום לTypeError-.

ניתן ליצור אובייקטים של wrapper objects, על ידי שימוש בבנאים:

String(), Number(),Boolean()

**אובייקטים גלובליים (Object Global):**

האובייקט global הוא אובייקט JavaScript רגיל המשרת מטרה חשובה מאוד: המאפיינים של אובייקט זה הם globally defined symbols הזמינים לתוכנית JavaScript.

כאשר ה- interpreter של JavaScript טוען דף חדש, הוא יוצר אובייקט גלובלי חדש ומעניק לו קבוצה ראשונית של מאפיינים המגדירים:

* מאפיינים גלובליים כמו undefined, Infinity, ו NaN
* פונקציות גלובליות כמו isNaN parseInt ו- eval
* בונה פונקציות כמו Date(), RegExp(), String(), Object(),Array()
* אובייקטים גלובליים כמו Math ו- JSON

ב- JavaScript בצד הלקוח, אובייקט Window משמש כאובייקט גלובלי עבור כל קוד JavaScript הכלול בחלון הדפדפן שהוא מייצג.

במילים פשוטות: אובייקטים הקיימים בשפה וזמינים בכל תוכנית JS.

**אתחול אובייקט ומערכים (object and array initializer):**

array initializer הוא רשימה מופרדת בפסיקים של ביטויים הכלולים בסוגריים מרובעים. לדוגמה:

var arr = []; //empty array: no values inside brackets means no elements

var mat = [[1, 2, 3], [4, 5, 6], [7, 8, 9]];

אלמנטים המכילים undefined , יכולים להיכלל באתחול מערך על ידי array literal באמצעות השמטת ערך בין שתי פסיקים.

לדוגמה, המערך הבא מכיל חמישה אלמנטים, כולל תאים בעלי הערך undefined:

var arr = [1, , , , 5];

בנוסף, ניתן להגדיר מערך בדרך הבאה:

var arr = new Array(10)

להלן דוגמה מלאה:

var arr1 = new Array(10); // 10 cells, each contains undefined.

var arr2 = new Array(10, 20, 30); // 3 cells: 10, 20, 30.

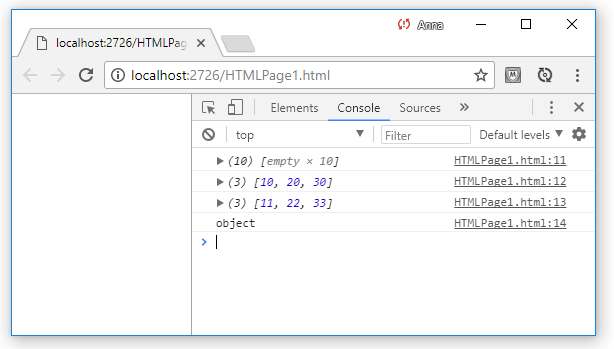
var arr3 = [11, 22, 33]; // 3 cells: 11, 22, 33.

console.log(arr1);

console.log(arr2);

console.log(arr3);

console.log(typeof arr1);



**הערה**: ניתן להוסיף למערכים איברים בצורה דינמית.

Object initializer הוא כמו array initializer, אך הסוגריים המרובעים מוחלפים בסוגריים מסולסלים, וכל subexpression(ביטוי) מורכב ממפתח המאפיין + נקודתיים + ערך המאפיין. לדוגמה:

// An object with 2 properties

var p = { x: 2.3, y: -1.2 };

//An empty object with no properties

var q = {};

//add to q the same properties as p

q.x = 2.3;

q.y = -1.2;

console.log(p); //{x:2.3, y:-1.2}

console.log(q); //{x:2.3, y:-1.2}

console.log(p==q); //false

:**Var**

מאז היווסדה, ל- JavaScript הייתה דרך אחת להכריז על משתנים: var.

הצהרת משתנים באמצעות var, עובדת לפי עקרון ה- variable ופועלת כאילו המשתנים הוכרזו בראש ה- execution contextהנוכחי (פונקציה).

הדבר עלול לגרום להתנהגות לא אינטואיטיבית, כפי שניתן לראות בדוגמה הבאה:

function func() {

// Intended to write to a global variable named 'num1'.

num1 = 4;

if (num1 == 4) {

// This declaration is moved to the top,

//causing the first write to 'num1' to act on the local variable

//rather than a global one.

var num1 = 3;

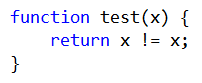
}

}

func();

console.log(num1); //should print 4 but results in an exception.

נתונה הפונקציה הבאה:



**איזה ערך יכול להישלח לפונקציה זו, בכדי שהיא תחזיר true? NaN**

**Let:**

הצהרת ה- let פועלת בדיוק כמו הצהרת ה- var, אך עם הבדל גדול: ההצהרות מוכרות רק בבלוק המקיף את המשתנה, וזמינות רק מהנקודה שבה ההצהרה ממוקמת.

לכן המשתנים המוצהרים על ידי let בתוך לולאה, או פשוט בתוך סוגריים מסולסלים, תקפים רק בתוך הבלוק הזה, ורק לאחר מכן תן הצהרה.

התנהגות זו היא הרבה יותר אינטואיטיבית. והשימוש ב let עדיף על השימוש ב- var ברוב המקרים.

כללים חשובים:

* **var** לא יכול להיות מוגדר פעמיים עם אותו שם בפונקציה אחת – (אפילו לא בלוק פנימי של הפונקציה) - למעשה, אנחנו יכולים להכריז פעמיים משתנה var, אבל זה לא יצור משתנה חדש, אלא עדיין יתייחס למשתנה בעל השם הזהה שהוגדר קודם לכן באותה הפונקציה.
* **let** - ניתן ליצור משתנה בעל שם זהה לבלוק החיצוני בתוך בלוק פנימי הגדרה זו תיצור משתנה חדש שיטיל צל על המשתנה החיצוני (אפקט ה- shadow).

**Const**:

הצהרת משתנה קבוע ב JavaScript -מתבצעת באמצעות המילה const.

כל הגדרת משתנה על ידי const מחייבת להשים ערך לתוך המשתנה בשורה בה הוא מוגדר.

הצהרת ה - const, בניגוד להצהרות ה - let וה- var, אינה מאפשרת לשנות את המשתנה לאחר ההצהרה הראשונית:

const obj = "Anna"; //obj is bound to the primitive string "Anna".

console.log(obj);

obj = [2, 4]; // TypeError

console.log(obj);

**const אינו הופך את הערך ל immutable**

Const פירושו שלמשתנה יש תמיד אותו ערך, אך אין פירוש הדבר שהערך עצמו הוא הופך להיות immutable (בלתי משתנה).

לדוגמה, obj הוא קבוע, אך הערך שהוא מצביע עליו הוא mutable (ניתן לשינוי) - ואנו יכולים להוסיף לו property (מאפיין):

const obj = {};

obj.x = 123;

console.log(obj.x); // 123

עם זאת, איננו יכולים לבצע השמה של ערך אחר לתוך המשתנה obj:

const obj = {};

obj = 123;

console.log(obj.x);

**פעולות על מערכים בJS –**

REVERSE - הופך את האלמנטים במערך.

POP - מסיר את האלמנט האחרון מהמערך ומחזיר אותו.

SHIFT – מסיר את האלמנט הראשון מהמערך ומחזיר אותו

PUSH – דוחף(מכניס) אלמנט נוסף מהסוף למערך , ומחזיר את הגודל החדש.

UNSHIFT – מכניס משתנה חדש לתחילת המערך ומחזיר את גודל המערך החדש.

LENGTH – מחזיר אורך מערך.

**Temporal dead zone**

למשתנה שהוכרז על ידי let או Const יש אזור מת זמנית (TDZ):

כאשר הקוד מגיע לבלוק המקיף של אותו משתנה (scope), לא ניתן לגשת אליו (לקבל את תוכן השתנה או להגדיר ערך למשתנה) עד שתבוצע השורה של ההצהרה.

בחלק הבא נבצע השוואה בין מחזורי החיים של משתני var (שאין להם TDZs) ומשתני let או Const (אשר יש TDZs).

**The life cycle of var-declared variables**

למשתני var אין temporal dead zones. מחזור החיים שלהם כולל את השלבים הבאים:

* כאשר הביצוע של הקוד מגיע ל scopeשל המשתנה(הפונקציה בה המשתנה מוגדר), מוקצה מיידית שטח אחסון (כולל binding) עבורו, והמשתנה מאותחל מיד, על ידי הערך undefined.
* כאשר הביצוע של הקוד בתוך ה scope מגיע להצהרה, המשתנה מקבל את הערך שצוין על ידי האתחול (assignment) - אם קיים.

אם אין initializer(אתחול), הערך של המשתנה נשאר undefined.

**The life cycle of let-declared variables**

משתנים שהוכרזו על ידיlet מכילים temporal dead zones ומחזור החיים שלהם הוא המחזור הבא:

* כאשר הביצוע של הקוד מגיע ל-scope (הבלוק המקיף אותו) של משתנה let, נוצר שטח אחסון (כולל binding) עבורו. המשתנה נשאר uninitialized (לא מאותחל).
* גישה למשתנה במצב uninitialized(לא מאותחל) גורמת ל - ReferenceError.
* כאשר הביצוע של הקוד בתוך ה- scope מגיע להצהרה, המשתנה מוגדר לערך שצוין על ידי האתחול (assignment) - אם קיים. אם לא קיים initializer אז הערך של המשתנה מוגדר ל undefined.

**משתני const** פועלים באופן דומה כדי למשתניlet , אבל הם חייבים להיות מאתחלים בשורת ההגדרה (כלומר, לקבל ערך מידי בשורת ההגדרה) ולא ניתן לשנות אותם בהמשך ה-scope.

**ערכי ברירת המחדל של פרמטרים ו- temporal dead zone**

אם לפרמטרים יש ערכי ברירת המחדל, הם נחשבים כמו רצף של הצהרות let והם כפופים ל temporal dead zones:

// OK: `y` accesses `x` after it has been declared

function func1(x = 1, y = x) {

return y;

}

console.log(func1()); // 1

// Exception: `x` tries to access `y` within TDZ

function bar(x = y, y = 1) {

return x;

}

console.log(func2()); // ReferenceError

אירועים ב JavaScript

**כמה דברים לפני:**

* שפת JavaScript הינה שפה "מונחת אירועים".
* "אירוע" הוא פעולה בדף ששפת ה- JavaScript יכולה לזהות ולתרגם כהוראה לביצוע.
* אלמנט- יכול להיות טקסט, כותרת, תמונה, תיבת טקסט...
* כל פעולה שהמשתמש מבצע על אלמנט בדף ה-HTML יוצר אירוע וכל אירוע כזה יכול לגרום להפעלת קוד.(למשל, נוכל לנצל את אירוע onclick לצורך הפעלת פונקציה ב JavaScript).
* אירועים עובדים לרוב בשיתוף עם פונקציות JavaScript, הפונקציות לא יופעלו אלא אם חל אירוע שמכוון להפעילן.

**להלן אירועים הקשורים לטעינת דף**:

1. **שם האירוע:** onload .

**הגורם להפעלת האירוע:** אירוע onload נוצר עם טעינת הדף.

1. **שם האירוע:** onClick .

**הגורם להפעלת האירוע:** האירוע onClick מתרחש כאשר המשתמש לוחץ על אלמנט.

**דוגמה: בעת לחיצה על הכפתור (נגיד יצרנו כפתור ("Click me" מופיע הפופ-אפ**.

1. **שם האירוע:** ondblclick

**הגורם להפעלת האירוע:** האירוע onblclick מתרחש כאשר המשתמש לוחץ לחיצה כפולה עם העכבר על אלמנט.

**דוגמה**: **בעת לחיצה כפולה עם העכבר על הכפתור מופיע הפופ-אפ.**

1. **שם האירוע:** onMousedown

**הגורם להפעלת האירוע:** האירוע onMousedown מתרחש כאשר משתמש מתחיל ללחוץ על לחצן העכבר מעל אלמנט – בעת ירידת לחצן העכבר.

**דוגמה**: **בעת לחיצה על העכבר (מיד כאשר מתחילים את הלחיצה) מופיע הפופ-אפ**.

1. **שם האירוע:** onMouseup

**הגורם להפעלת האירוע:** האירוע onMouseup מתרחש כאשר משתמש עוזב את לחצן העכבר בלחיצה על אלמנט – בעת עליית לחץ העכבר.

**דוגמה : בעת עזיבת הלחיצה על הכפתור מופיע הפופ-אפ**

1. **שם האירוע:** onMouseover

**הגורם להפעלת האירוע:** האירוע onMouseup מתרחש כאשר סמן העכבר נכנס לאזור האלמנט.

**דוגמה: בעת כניסת הסמן לאזור הכותרת מופיע הפופ-אפ.**

1. **שם האירוע:**  onMouseout

**הגורם להפעלת האירוע:** האירוע onMouseout מתרחש כאשר סמן העכבר יוצא מאזור האלמנט.

**דוגמה: בעת יציאת הסמן מאזור הכותרת מופיע הפופ-אפ.**

1. **שם האירוע:** onMousemove

**הגורם להפעלת האירוע:** האירוע onMousemove מתרחש כאשר סמן העכבר זז בזמן שהוא מעל אלמנט.

**דוגמה : בעת הזזת הסמן באזור הכותרת מופיע הפופ-אפ**

1. **שם האירוע:** onkeypress

**הגורם להפעלת האירוע:** האירוע onkeypress מתרחש כאשר המשתמש לוחץ על מקש כלשהו במקלדת.

**דוגמה**: **בעת הקשה על מקש במקלדת כאשר אנחנו בפוקוס בתוך התיבת טקסט מופיע הפופ-אפ**.

1. **שם האירוע:** onkeydown

**הגורם להפעלת האירוע:** האירוע onkeydown מתרחש כאשר המשתמש מוריד ע''י לחיצה מקש כלשהו במקלדת.

**דוגמה**: **בעת התחלת לחיצה על מקש במקלדת כאשר אנחנו בפוקוס בתוך התיבת טקסט מופיע הפופ-אפ**.

1. **שם האירוע:** onkeyup

**הגורם להפעלת האירוע:** האירוע onkeydown מתרחש כאשר המשתמש משחרר לחיצה מקש כלשהו במקלדת.

**דוגמה**: **בעת שחרור ההקשה על מקש במקלדת כאשר אנחנו בפוקוס בתוך התיבת טקסט מופיע הפופ-אפ.**

1. **שם האירוע:** onfocus

**הגורם להפעלת האירוע:** האירוע onfocus מתרחש כאשר תיבת קלט מקבלת פוקוס, אירוע זה נוצר עם כניסת הסמן אל השדה ע''י לחיצת העכבר בשדה או ע''י מעבר באמצעות טאב.

**דוגמה: בעת כניסה לתיבת הקלט ע"י לחיצה עם העכבר בתוכה (יוצר פוקוס) מופיע הפופ-אפ.**

1. **שם האירוע:** onblur

**הגורם להפעלת האירוע:** האירוע onblur מתרחש כאשר אלמנט של תיבת קלט מאבד את הפוקוס.

**דוגמה: בעת יציאה מתיבת הקלט ע"י לחיצה עם העכבר מחוץ לאותה תיבה (יוצר איבוד פוקוס) מופיע הפופ-אפ.**

1. **שם האירוע:** onchange

**הגורם להפעלת האירוע:** האירוע onchange נוצר בעת שינוי תוכן השדה. בעת יציאה מהשדה לאחר הכתיבה בו.

**דוגמה : בעת יציאה מתיבת הקלט ע"י לחיצה עם העכבר מחוץ לאותה תיבה, לאחר שהקלדנו טקסט בתוך התיבה. מופיע הפופ-אפ.**

**סינכרוני וא-סינכרוני הבדלים –**

קוד שפועל בצורה סינכרונית ירוץ שורה אחרי שורה אחרי שורה כמו רובוט, יחכה עד שהשורה תסתיים ורק אז ימשיך לשורה הבאה.

קוד שפועל בצורה **אסינכרונית** ירוץ שורה אחרי שורה גם כן, אך כאשר יגיע לשורת קוד בה יאלץ להמתין לדוגמה SETTIMEOUT אז הוא לא יעצור ויחכה אלא ימשיך לפעולות הבאות ושיגמר הTIMEOUT יחזור אליו ויבצע את מה שצריך.

**איך יכול להיות שJS היא שפה אסינכרונית וגם רק ONE THREADED ?**

שפת JS היא שפה של ONE THREAD והתנהגות א-סינכרונית היא לא חלק רגיל מהשפה, אלא התנהגות מובנת בתוך הליבה של הJS בדפדפן .

**מהו EVENT LOOP ?**

הרעיון המרכזי בו הוא תכנות מונחה עצמים בצד שרת. הקוד יהיה מבוסס אירועים כמו קוד שרץ בדפדפן.

מכיוון שלא יעיל בכלל ומאוד איטי לחכות לתשובה מכל בקשה ורק אז להמשיך לשורת הקוד הבאה, נוצר הEVENT LOOP, שמאפשר לתוכנית להמשיך לרוץ, ולחזור לEVENT LOOP בצורה מיידית, כאשר הEVENT נגמר.

**ישנם 3 מקרים ב- js שאנו משתמשים ב callback לשימוש בקוד אסינכרוני:**

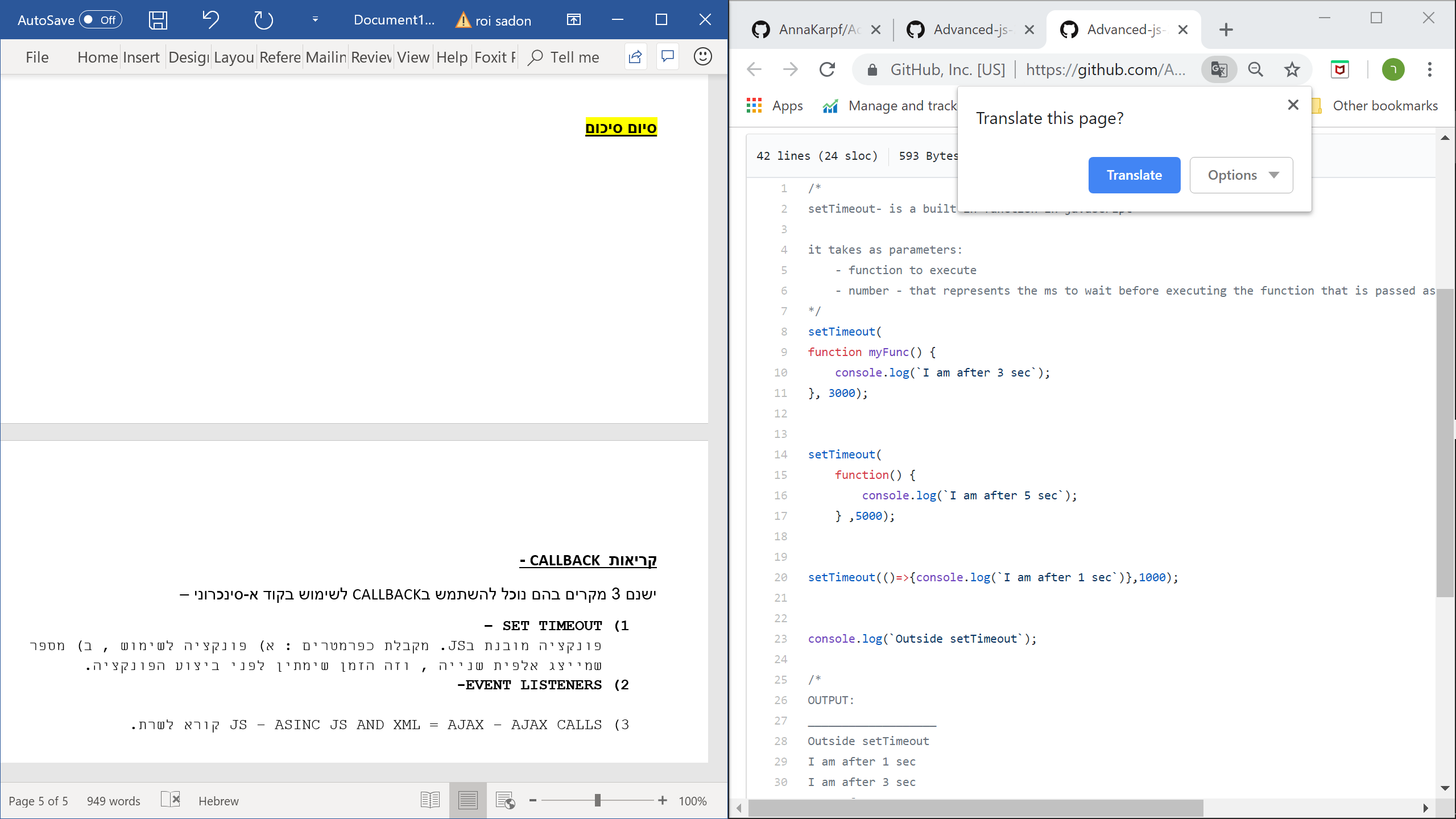
# There are 3 cases in js that we use callback to use async code:

* setTimeout
* Event listeners
* Ajax calls - Ajax = Async Js And Xml - js calls to the server

**דרכים לכתיבת קוד א-סינכרוני:**

**קריאות CALLBACK -**

ישנם 3 מקרים בהם נוכל להשתמש בCALLBACK לשימוש בקוד א-סינכרוני –

1. **SET TIMEOUT –**

פונקציה מובנת בJS.

מקבלת כפרמטרים :

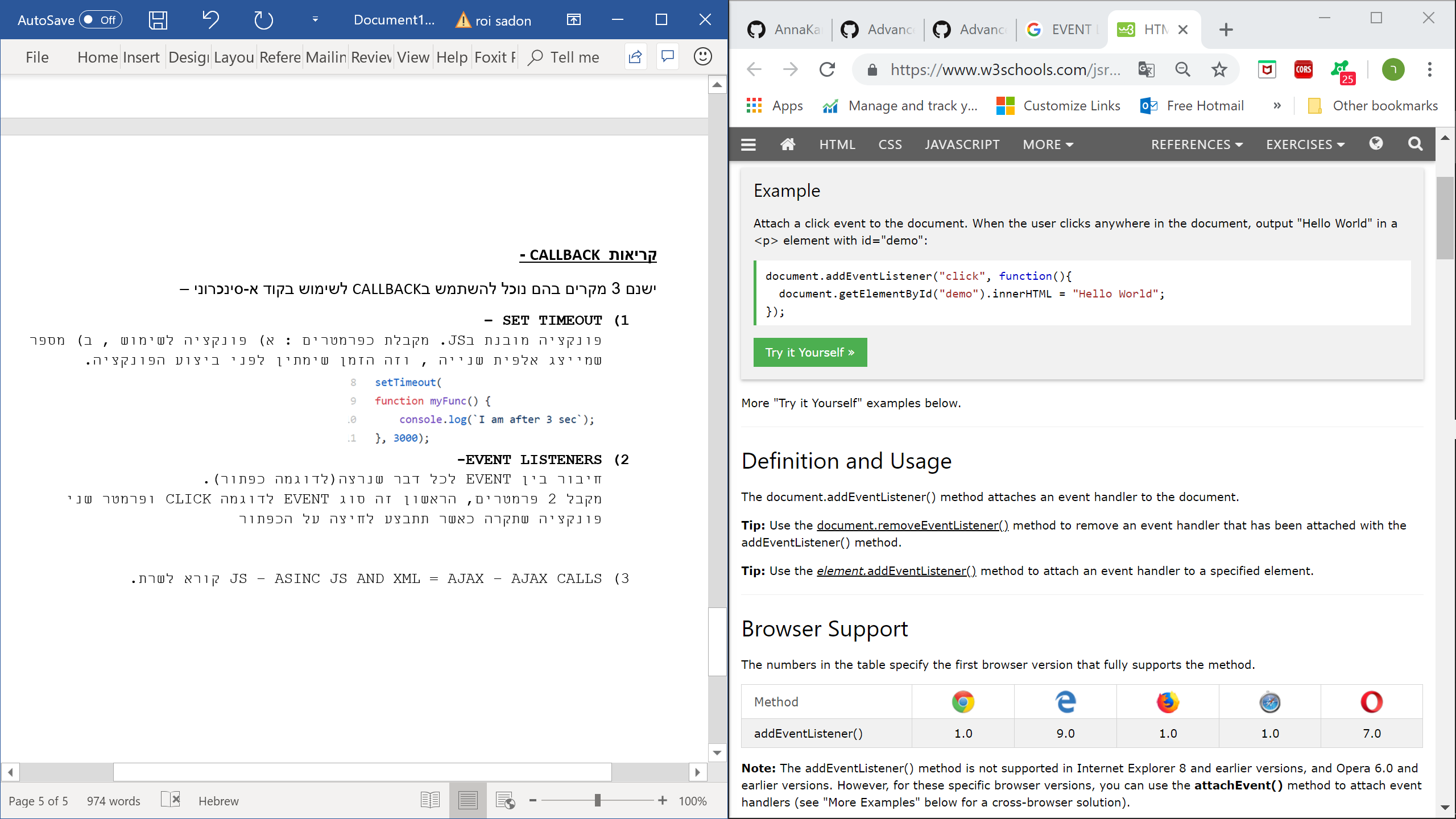
א) פונקציה לשימוש

ב) מספר שמייצג אלפית שנייה , וזה הזמן שימתין לפני ביצוע הפונקציה.

1. **EVENT LISTENERS-**

חיבור בין EVENT לכל דבר שנרצה(לדוגמה כפתור).

מקבל 2 פרמטרים, הראשון זה סוג EVENT לדוגמה CLICK ופרמטר שני פונקציה שתקרה כאשר תתבצע לחיצה על הכפתור



1. **AJAX CALLS – AJAX = ASINC JS AND XML –**

שליחת קריאה בJS לשרת.

**HTTP REQUESTS – AJAX –**

**XML HTTP REQUEST –** יצטרך 2 "מאזינים" שיוגדרו כדי לקלוט את המידע שחוזר ולטפל במקרי הERROR ובמקרי SUCCESS יכיל גם OPEN() ו – SEND().

**FETCH –** מאפשר ליצור בקשות תקשורת דומות ל XMLHttpRequest , ההבדל העיקרי הוא שFETCH משתמש בPROMISES , מה שיצור ייעול של הקוד , ונמנעים מעומס שיווצר ע"י מספר קריאות CALLBACK.

**PROMISE –**

בגדול זוהי פשוט דרך נוחה לתפוס אירועים א-סינכרונים ולהגיב אליהם.

זהו אובייקט שמקבל פונקציה עם 2 פרמטרים REJECT ו RESOLVE. בתוך PROMISE ניתן לעשות **הכל**  , וכשנסיים נחזיר RESOLVE אם הכל עבד או REJECT במידה וכשל.

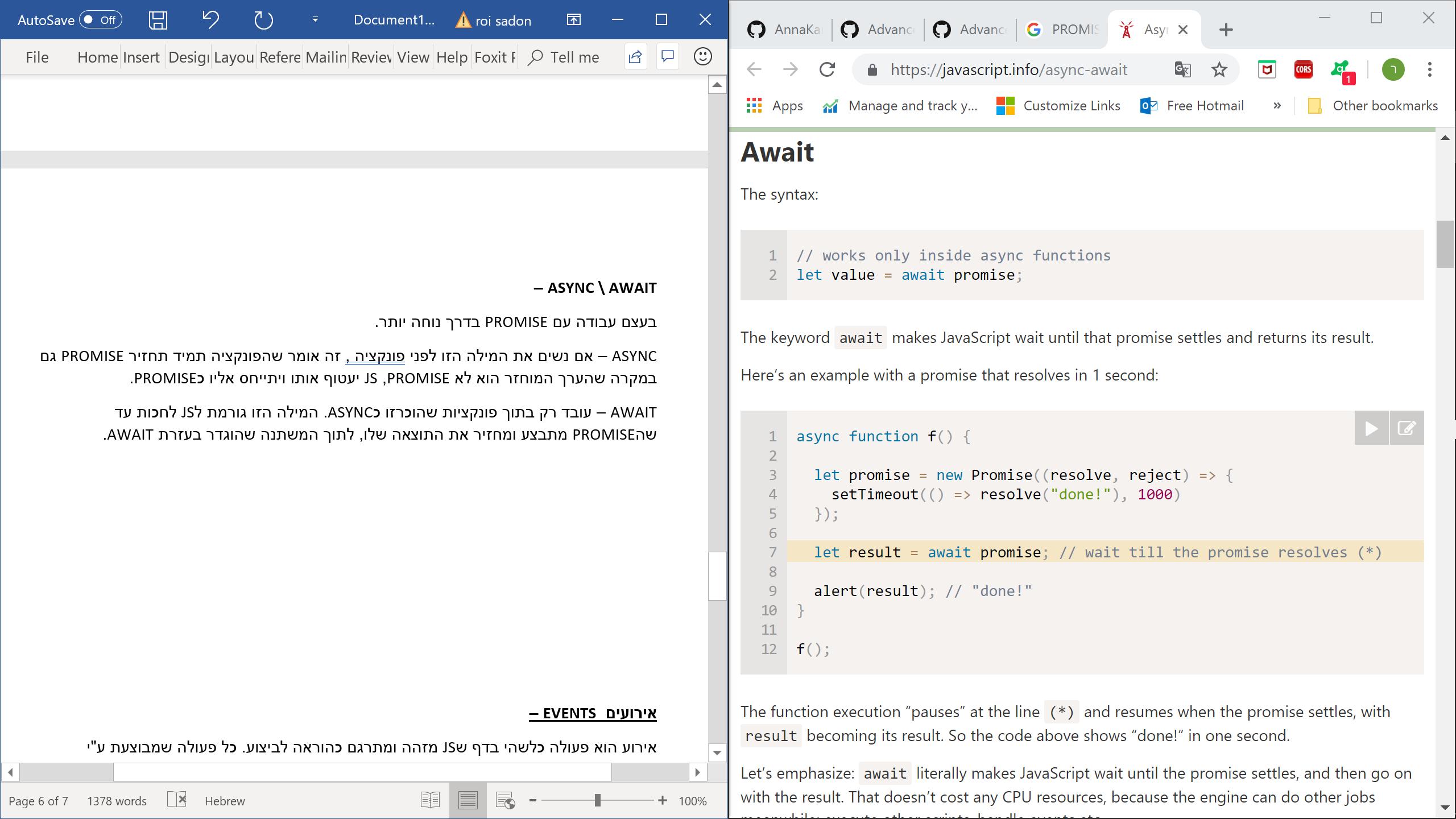
PROMISE מחולק ל 2 חלקים – החלק הראשון מכיל את הPROMISE עצמו עם הפונקציה שמקבלת פרמטרים. החלק השני הוא החלק של THEN וCATCH– שמקבלים 2 פרמטרים :

1) פונקציה אנונימית שרצה כאשר יש לנו RESOLVE

2) פונקציה שרצה כאשר יש לנו REJECT.

**ASYNC \ AWAIT –**

בעצם עבודה עם PROMISE בדרך נוחה יותר.

ASYNC – אם נשים את המילה הזו לפני פונקציה , זה אומר שהפונקציה תמיד תחזיר PROMISE גם במקרה שהערך המוחזר הוא לא PROMISE, JS יעטוף אותו ויתייחס אליו כPROMISE.

AWAIT – עובד רק בתוך פונקציות שהוכרזו כASYNC.

המילה הזו גורמת לJS לחכות עד שהPROMISE מתבצע

ומחזיר את התוצאה שלו, לתוך המשתנה שהוגדר בעזרת

AWAIT.