

Advanced JavaScript

Advanced JavaScript

Including es5+es6

JOHN BRYCE

تلמוד ה-טק. זה לעבוד!

a matrix company

תוכן עניינים

5.....	1. מבוא.....
7.....	2. סוגים משתנים - Data types
7.....	2.1. טיפוס משתנה דינמי
8.....	2.2. סוג הטיפוסים
9.....	2.3. Ref type VS value type
12.....	== VS === 2.4.
13.....	2.5. אופרטור typeof
17.....	2.6. טיפוס number
18.....	2.7. טיפוס string
22.....	2.8. טיפוס boolean
22.....	2.9. טיפוס undefined - או null
23.....	2.10. Wrapper Objects
24.....	2.11. global object
25.....	2.12. Object and array initializers
29.....	3. הגדרת משתנים – Variable Declaration
29.....	3.1. VAR
30.....	3.2. let
38.....	3.3. Const
42.....	3.4. Temporal dead zone
48.....	3.5. לולאות והגדרת משתנים
53.....	3.6. סיכון אופני הגדרת משתנים
54.....	4. פונקציות
54.....	4.1. דרכי להגדרת פונקציות

56.....	function hoisting 4.2.
56.....	Nested functions 4.3.
57.....	Function overloading 4.4.
59.....	Arrow functions 4.5.
63.....	הגדרת משתנים גלובליים בפונקציה 4.6.
65.....	Self-involve functions 4.7.
66.....	Closures 4.8.
69.....	Arguments and parameters 4.9.
70.....	Invoking functions 4.10.
79.....	תרגילים 4.11.
82.....	5. אובייקטים
82.....	מבנה האובייקט 5.1.
83.....	יצירת אובייקט 5.2.
88.....	קריאה המאפיינים ו שינוי האובייקט 5.3.
98.....	6. אסינכריוני
99.....	Callback function 6.1.
101.....	Promises 6.2.
103.....	Async await 6.3.
107.....	7. מחלקות וורשה ו-prototype 7.
107.....	Constructor function 7.1
115.....	class 7.2
124.....	תרגילים 7.3.

1. מבוא

,Netscape Communications Corporation, מפתח ב- Brendan Eich, מפתח על ידי European Computer Manufacturers Association – התאחדות אירופאית לייצרני מחשבים. ארגון ללא מטרות רווח ש矜持ה היא לקבוע סטנדרטים לתקשות. הסטנדרט המוכר ביותר של הארגון הוא.

ECMAScript היא ספציפיקציה לשפת סקריפט. במקור ECMA הוא ראשי תיבות של European Computer Manufacturers Association – השם המקורי של הארגון הוא. ECMAScript – קלומרת JavaScript הבנויה על בסיס הסטנדרטים של ECMAScript – קלומרת ECMAScript הינה הוראות של פונקציונליות, תוספות פונקציונליות, תיקוני באגים, ויכולות מתקדמות שפות המרחיבות את הספציפיקציה.

JavaScript מתחדשת בתקנים חדשים של שינויים, תוספות פונקציונליות, תיקוני באגים, ויכולות מתקדמות חדשות המרחיבות את הספציפיקציה.

להלן רשימה מסכמת של הגרסאות עד לשנת 2017:

- 1997 - ECMAScript 1
 - First Edition.
- 1998 - ECMAScript 2
 - Editorial changes only.
- 1999 - ECMAScript 3
 - Regular expressions
 - The do-while block
 - Exceptions and the try/catch blocks
 - More built-in functions for strings and arrays
 - Formatting for numeric output
 - The in and instanceof operators
- ECMAScript 4
 - Was never released.
- 2009 - ECMAScript 5
 - Added "strict mode".
 - Added JSON support.
- 2011 - ECMAScript 5.1

- Editorial changes.
- 2015 - ECMAScript 6- ECMAScript 2015.
 - Added classes and modules.
- 2016 - ECMAScript 7- ECMAScript 2016
 - Added exponential operator (**).
 - Added Array.prototype.includes.

הוסיפו פיצ'רים רבים לשפה, ביניהם:

- Support for constants
- Block Scope
- Arrow Functions
- Extended Parameter Handling
- Template Literals
- Extended Literals
- Enhanced Object Properties
- De-structuring Assignment
- Modules
- Classes
- Iterators
- Generators
- Collections
- New built in methods for various classes
- Promises

שימוש לב, עדין לא כל הדפנסים הטמינו את יכולות להריץ את הקוד בתחביר החדש, אולם מכיוון שהשימוש בה נפוץ מאוד, ומאפשר ליצור קוד JavaScript קרייא יותר וקל לתחזוק, רבים מהמתכננים מעדיפים בכל זאת להשתמש בגרסאות החדשנות של- ECMAScript. אפשר להשתמש בכך גם ב- Babel שלוקח קוד שלJavaScript כתוב לפי התקן החדש ביותר וועשה לו טרנספיילינג (המרה של קוד אחד בקוד אחר בהתאם רמת אבסטרקציה) ומשלב בתוכו שכתוב של הקוד כך שיתאים לתקנים ישנים יותר.



2. סוגים משתנים - Data types

1.2. טיפוס משתנה דינמי |

JavaScript היא שפת תכנות דינמית, ולכן אין צורך להזכיר על סוג של משתנה בזמן ההצהרה. סוג המשתנה יקבע באופן דינמי בזמן ביצוע התוכנית (זמן ריצה).

בעקבות זאת, JavaScript מאפשרת לאותו משתנה לקבל ערכים מטיפוסים שונים. כדי שניתן לראות בדוגמה הבאה:

```
<!DOCTYPE html>

<html lang="en" xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta charset="utf-8" />
    <title></title>
    <script>

        var x;

        //x has number type
        x = 42;

        //x has string type
        x = "John bryce";

        //x has boolean type
        x = true;

        //x has object type
        x = { firstName: 'Anna', lastName: 'Karp' };

    </script>

</head>
<body>
</body>
</html>
```

2.2. סוג הטיפוסים

ניתן לחלק לשתי קטגוריות את סוגי הטיפוסים ב- JavaScript:

1. primitive types:

- string
- boolean
- number
- null
- undefined
- symbol (new in es6)

2. object types:

- כל טיפוס שלא נכלל בטיפוסים הפרימיטיביים, נכלל תחת object types.
- אובייקטים מובנים של JavaScript הנפוצים בשימוש, הם המחלקה Date המגדירה אובייקטים המיצגים תאריכים. המחלקה RegExp מגדירה אובייקטים המיצגים מusters (pattern-matching).
- והמחלקה Error המגדירה אובייקטים המיצגים שגיאות זמן ריצה שאפשרו להתרחש בתוכנת JavaScript.
- אובייקט הוא אוסף של מאפיינים שבهم לכל מאפיין יש שם וערך (או ערך פרימיטיבי כגון מספר, או אובייקט).
- אובייקט JavaScript רגיל, הוא unordered collection של מפתחות וערכים.
- JavaScript מגדירה גם סוג ordered collection, הידוע כמערך (Array) המציג אוסף מסודר שלערכים מסווגים ע"י אינדקס מספרי ולא ע"י מפתח מחוץ-
- פונקציה מוגדרת ב- JavaScript כתיפוס של אובייקט.

ה-**interpreter** של JavaScript מבצע automatic garbage collection ניהול זיכרון. משמעות הדבר היא כי תוכנית יכולה ליצור אובייקטים לפי הצורך, והתוכנת אף פעם לא צריכה לבצע deallocation עברו האובייקטים שנוצרו. כאשר אובייקט אינו נגיש יותר (لتוכנית אין עוד דרך לגשת אליו) ה-**interpreter** יידע שלא ניתן להשתמש באותו אובייקט שוב, ומשחרר באופן אוטומטי את הזיכרון שהוקצה עבור האובייקט.

mutable and immutable types:

- **mutable** - ערך של משתנה מסוג mutable יכול להשתנות קבוע האובייקטים הם mutable
 - **immutable** - number, boolean, null, undefined string ואנו, אינם ניתנים לשינוי.
- ניתן לגשת לტקסט בכל אינדקס של מחזורת, אך JavaScript אינה מספקת אפשרות לשנות את הtekst של מחזורת קיימת.

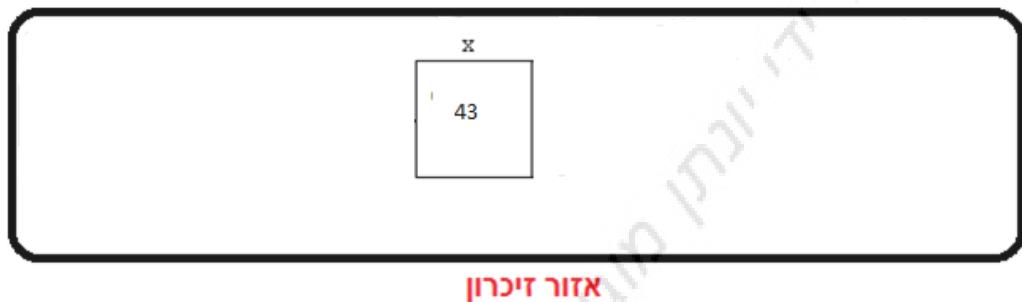
Ref type VS value type . 2.3

- המשתנים מסווג **value type** הם משתנים פרימיטיביים (כגון: number, boolean, string) וכאשר המשתנה נוצר בזיכרון, הוא יכול בתוך שטח המשתנה עצמו את הערך המושם לתוכו.

לדוגמא, כאשר ניצור את ההגדירה הבאה:

```
var x=43;
```

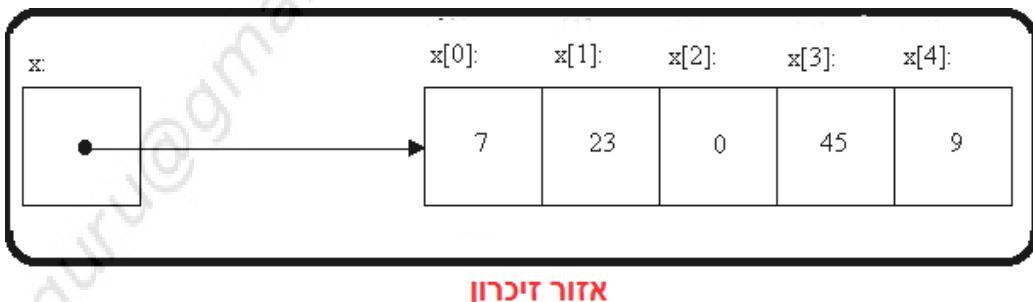
בזיכרון יוצר המצביע הבא:



- המשתנים מסווג **reference type** הם משתנים מטיפוס object type, והמשתנה מכיל הפניה לשטח שבו האובייקט נוצר.
לדוגמא, עבור ההגדירה:

```
var x=[7,23,0,45,9];
```

נקבל בזיכרון את המפה הבא:



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

- .1. בביטוי השוואה בין שני משתנים (`==`):
במשתני **value type** – "יבדק האם שני המשתנים מכילים אותו ערך".

- במשתני type – "יבדק האם שני המשתנים מכילים הפניה לאותו אובייקט.

לדוגמה, הקוד הבא:

```
<!DOCTYPE html>

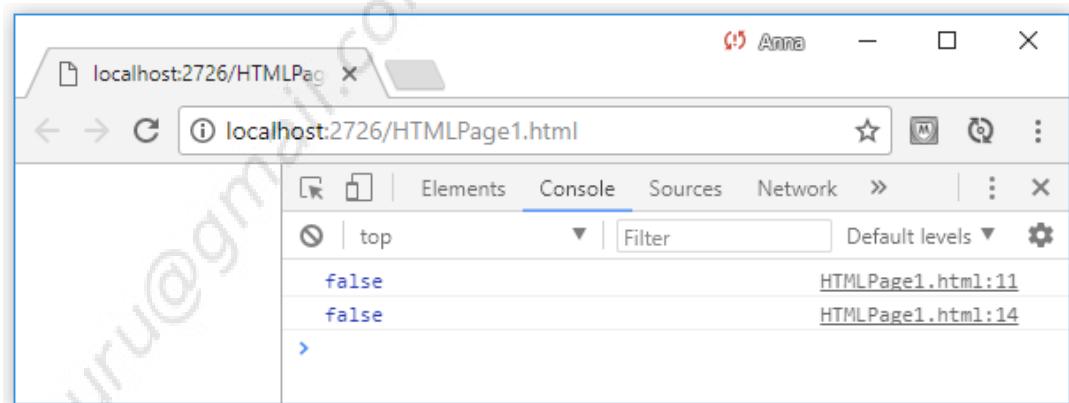
<html lang="en" xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta charset="utf-8" />
    <title></title>

    <script>

        var o = { x: 1 }, p = { x: 1 };      //Two objects with the same properties
        console.log(o === p);               //false

        var a = [], b = [];                 //Two distinct, empty arrays
        console.log(a === b);               //false
    </body>
</html>
```

כאשר נורץ את הקוד בדף, נקבל את התוצאה הבאה:



1. ביצוע השמה של משתנה אחד לתוך משתנה אחר:
 - במשתנה מסוג value type – תבצע העתקת הנתון שבתוך המשתנה
 - במשתנה מסוג reference type – תבצע העתקת הפניה אליה מצביע המשתנה

כאשר נבצע שינוי על העותק של משתנה פרימיטיבי, משתנה המקור לא יושפע.

ואילו אם נבצע שינוי על העותק של משתנה לא פרימיטיבי, משתנה המקור יושפעו כיון שהוא מצביע לאותו אובייקט שתוכנו שהעותק מצביע.

לדוגמה, ניצור מערך בתוך משתנה a, נעתיק את תוכן a ל-b, ונראה שככל שינוי שביצעו על b יופיע גם ב-a:

```
<!DOCTYPE html>

<html lang="en" xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta charset="utf-8" />
    <title></title>
    <script>
        var a = [];      // The variable      a refers to      an empty array.
        var b = a;      // Now b refers      to the same array.

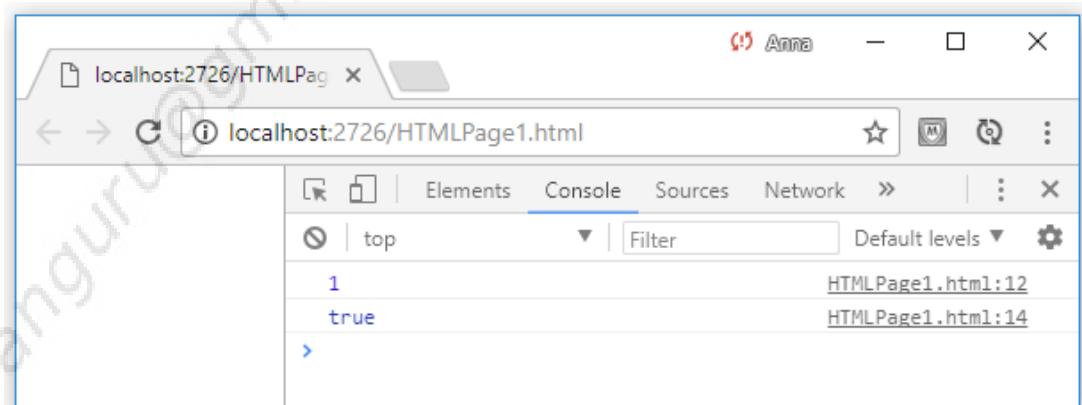
        b[0] = 1;      // Mutate the array      referred to by variable      b.
        console.log(a[0]);    //1 (the change is      also visible through variable a)

        console.log(a === b); //true (a and b refer to the same object)

    </script>
</head>
<body>

</body>
</html>
```

כאשר נריץ את הקוד בדפדפן, נקבל את התוצאה הבאה:



dagsh: בשליחת פרמטרים בפונקציה, מתחבעה העתקה של הפרמטר שנשלח בקריאה לפונקציה, אל הפרמטר שמתקבל על ידי הפונקציה, וכך ישים שני סוגי העברות פרמטרים:

- 1) העברת פרמטר **by value** – תבוצע כאשר נעביר משתנים פרימיטיביים.
קוד שקורא לפונקציה ושולח לה ערך של תוכן משתנה שתוכנו מועתק לתוך הפרמטר המקומי של הפונקציה.
- 2) העברת פרמטר **by reference** – תבוצע כאשר נעביר לפונקציה משתנה שהוא לא פרימיטיבי.
הקוד שקורא לפונקציה שולח לה ערך של כתובות למשנה מסוים בזיכרון, והכתובות מועתקת לתוך הפרמטר המקומי של הפונקציה.

== VS === . 2.4 |

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

לדוגמה, הקוד הבא:

```
<!DOCTYPE html>

<html lang="en" xmlns="http://www.w3.org/1999/xhtml">
<head>
  <meta charset="utf-8" />
  <title></title>
  <script>
    console.log("0 == ''", 0 == ''); // true
    console.log("0 === ''", 0 === ''); // false

    console.log("0 == '0'", 0 == '0'); // true
    console.log("0 === '0'", 0 === '0'); // false

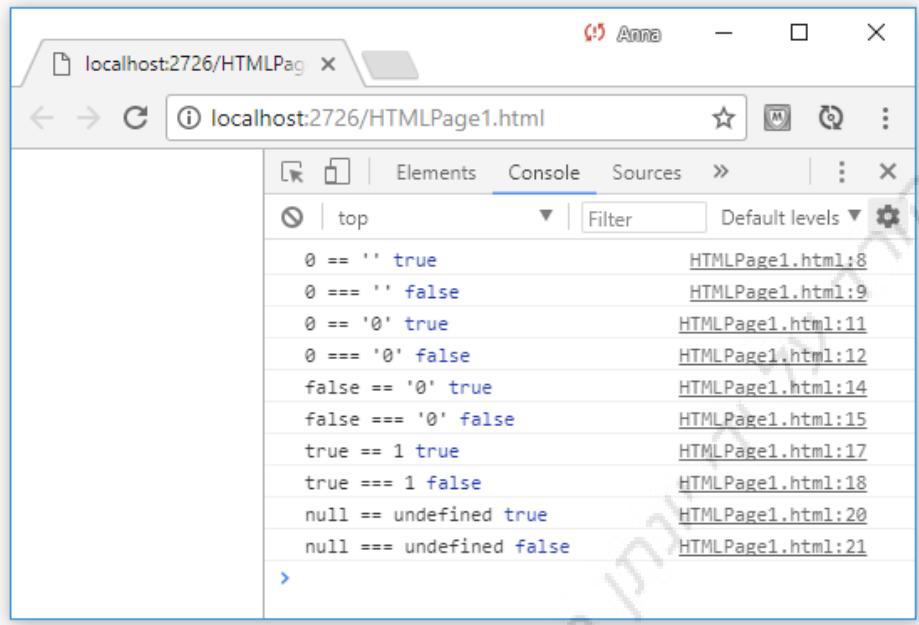
    console.log("false == '0'", false == '0'); // true
    console.log("false === '0'", false === '0'); // false

    console.log("true == 1", true == 1); // true
    console.log("true === 1", true === 1); // false

    console.log("null == undefined", null == undefined); // true
    console.log("null === undefined", null === undefined); // false
  </script>
</head>
<body>

</body>
</html>
```

כאשר נרץ את הקוד בדפדפן, נקבל את התוצאה הבאה:



2.5. אופרטור typeof |

אופרטור typeof משמש כדי למצוא את סוג המשתנה.

לדוגמה, הקוד הבא:

```

<!DOCTYPE html>

<html lang="en" xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta charset="utf-8" />
    <title></title>

    <script>
        //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()));
    </script>

```

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

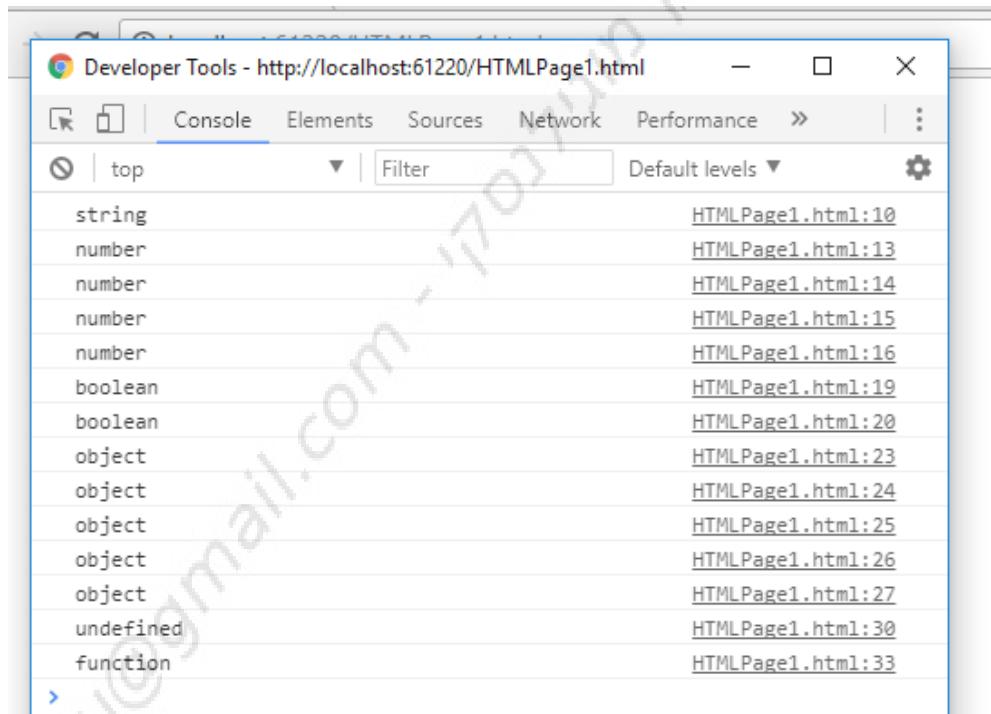
//typeof undefined
console.log(typeof undefined);

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

</script>
</head>
<body>

</body>
</html>
```

כאשר נרץ את הקוד בדפדפן, נקבל את התוצאה הבאה:



הבדל בין === ל-typeof

לעתים קרובות נרצה לבדוק האם משתנה מסוים מכיל את הערך `undefined`. ניתן לבצע את הבדיקה במספר דרכים:

• בדיקה באמצעות `==`

לדוגמה, הקוד הבא:

© כל הזכויות שמורות לジョン ברייס הדרכה בע"מ מקבוצת מטריקס

```
<!DOCTYPE html>

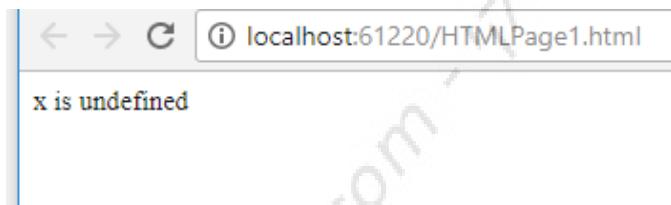
<html lang="en" xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta charset="utf-8" />
    <title></title>
    <script>

        var x;

        if (x === undefined) {
            document.write("x is undefined");
        }
        else {
            document.write("x is defined")
        }
    </script>

</head>
<body>
</body>
</html>
```

כasher נרץ את הקוד בדף, נקבל את התוצאה הבאה:



• בדיקה באמצעות **typeof**

לדוגמה, הקוד הבא:

```
<!DOCTYPE html>

<html lang="en" xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta charset="utf-8" />
    <title></title>
    <script>

        if (typeof x === 'undefined') {
            document.write("x is undefined");
        }
        else {
            document.write("x is defined")
        }
    </script>
```

```
if (x === undefined) {
    document.write("x is undefined");
}
else {
    document.write("x is defined")
}

</script>

</head>
<body>
</body>
</html>
```

כasher נרץ את הקוד בדפסן, נקבל את התוצאה הבאה:

The screenshot shows a browser developer tools window with the title "Elements Sources Network Performance >". Below the tabs, it says "Serving from the file system? Add your files into the ... more never show x". The main area displays the following HTML code:

```
<!DOCTYPE html>
<html lang="en" xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta charset="utf-8" />
<title></title>
<script>
if (typeof x === 'undefined') {
    document.write("x is undefined");
}
else {
    document.write("x is defined")
}

</script>
</head>
<body>
</body>
</html>
```

A tooltip box appears over the line "if (x === undefined){", containing the text "Uncaught ReferenceError: x is not defined".

סיכום:



ty יכול גם לעבוד על משתנה undefined typeof
 == undefined יזרוק ReferenceError עבור שהמשתנה

-
-

2.6. טיפוס number |

JavaScript אינה עושה הבחנה בין ערכים שלמים וערכים ממשיים.
כל המספרים ב- JavaScript מיוצגים באמצעות פורמט נקודה צפה 64-bit המוגדר על ידי תקן IEEE 754.

number literals

- מספרים שלמים
לדוגמה: 123

ערכים הקסדצימליים (בסיס 16)
hexadecimal literal מתחילה עם 0x או X0 ואחריו מספר הקסדצימלי.
ספרה הקסדצימלית מייצגת ערכים בטוחה 0 עד 15, ויכולת להיות מיוצגת באמצעות אחת המערכים הבאים:

- 0-9
- A-F
- a-f

לדוגמה: 0xff0

- Floating-point literals – יכולם להיות מספר המכיל decimal point
לדוגמה: 3.14

ניתן לייצג גם את האותיות העכשוויות באמצעות exponential notation: מספר ממשי ואחריו האות E או e.
לדוגמה: 23e6.02

אפשרוני להוסיף סימן פלאס או מינוס, ואחריו מעריךשלם.
לדוגמה: 3-E1.473

2.7. טיפוס string

מחרוזת היא רצף מסודר של תווים, כאשר כל תו מורכב מ 16 סיביות.
אורך מחרוזת הוא מספר הערכות של 16 סיביות שהוא מכיל.

- JavaScript אין סוג מיוחד המיצגתו אחד של מחרוזת. וכך ליצג ערךתו ייחד יש להשתמש במחרוזת בעלת אורך של תו אחד.

ב-ECMAScript ניתן לטפל במחרוזות כמו `read-only arrays`, ויש אפשרות לגשת לתווים בודדים של מחרוזת באמצעות סוגרים מרובעים במקומם בשיטת `() charAt`

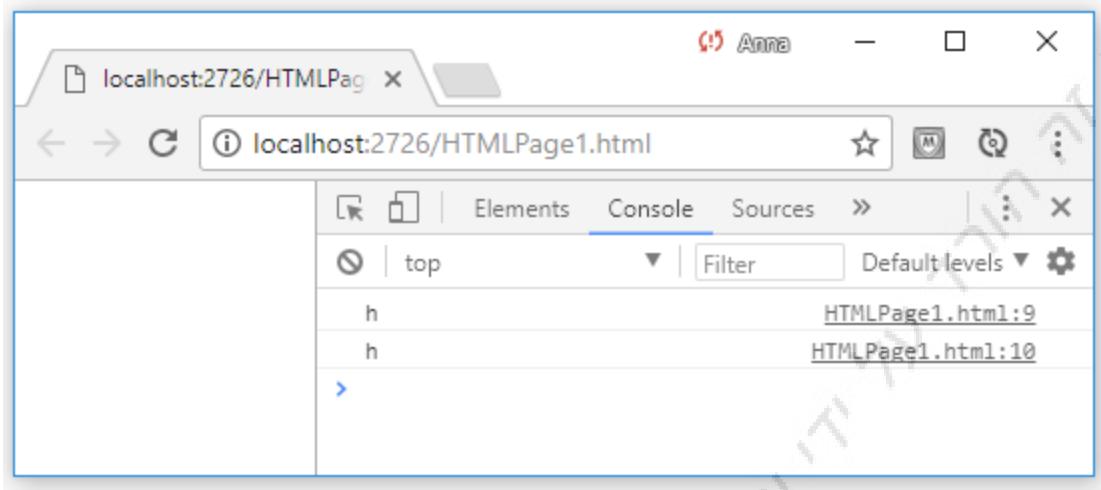
לדוגמה:

```
<!DOCTYPE html>

<html lang="en" xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta charset="utf-8" />
    <title></title>
    <script>
        var s = "hello, world";
        console.log(s[0]);
        console.log(s.charAt(0));
    </script>
</head>
<body>

</body>
</html>
```

כאשר נריץ את הקוד בדף, נקבל את התוצאה הבאה:



JavaScript Template String Literals

בכל פרויקט יהיו שימושים רבים באינטראפלציה על מנת לשתול ערכים לתוך מחרוזת.
הדרך הסטנדרטית לעשות זאת ב- JavaScript היא באמצעות : repeated concatenations

```
<!DOCTYPE html>

<html lang="en" xmlns="http://www.w3.org/1999/xhtml">
<head>
  <meta charset="utf-8" />
  <title></title>
  <script>

    var firstName="Anna";
    var lastName = "Karp";

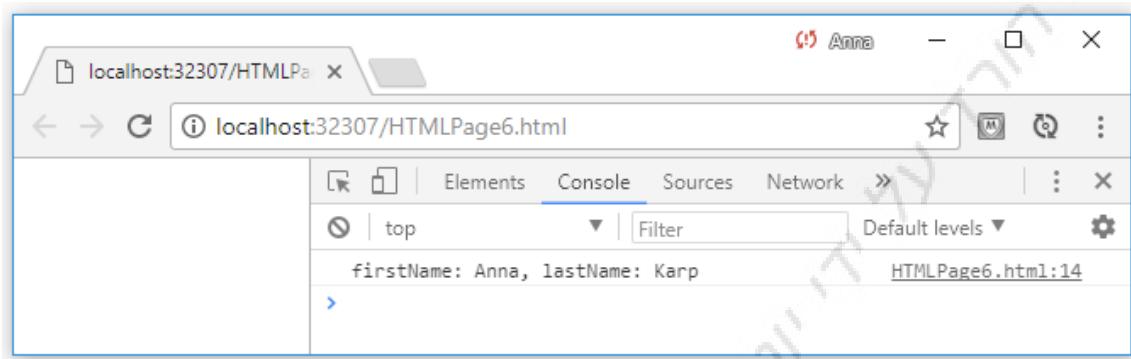
    var str = 'firstName: ' + firstName + ', lastName: ' + lastName;

    console.log(str);

  </script>
</head>
<body>

</body>
</html>
```

כאשר נרץ את הקוד בדף, נקבל את התוצאה הבאה:



מביא פתרון הרובה יותר אינטואיטיבי וקל לשימוש:

```
<!DOCTYPE html>

<html lang="en" xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta charset="utf-8" />
    <title></title>
    <script>

        var firstName="Anna";
        var lastName = "Karp";

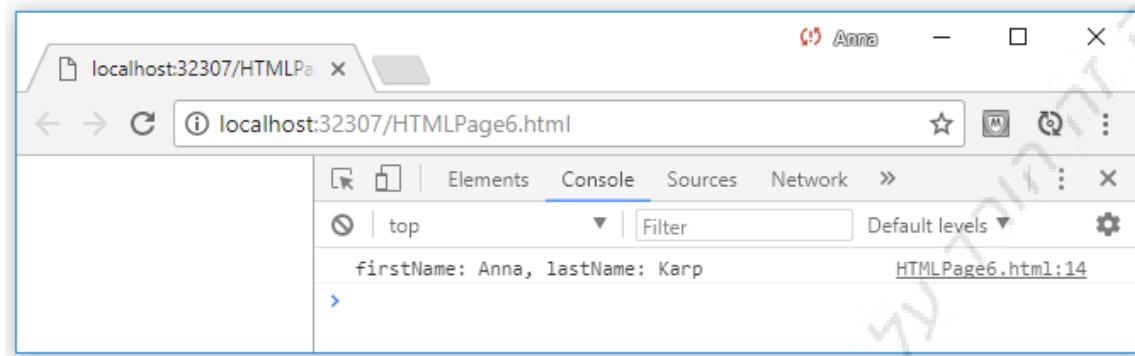
        var str = `firstName: ${firstName}, lastName: ${lastName}`;

        console.log(str);

    </script>
</head>
<body>

</body>
</html>
```

כאשר נרץ את הקוד בדף, נקבל את התוצאה הבאה:



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

```
<!DOCTYPE html>

<html lang="en" xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta charset="utf-8" />
    <title></title>
    <script>

        var firstName="Anna";
        var lastName = "Karp";

        var str = `

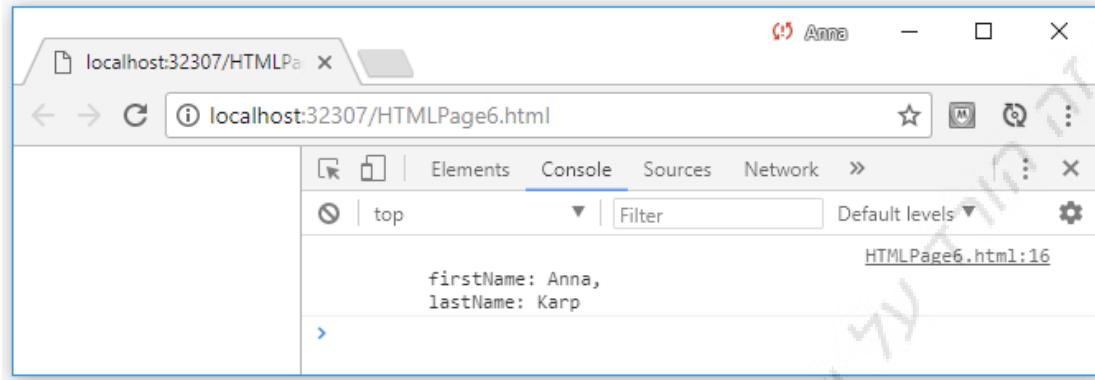
            firstName: ${firstName},
            lastName: ${lastName}`;

        console.log(str);

    </script>
</head>
<body>

</body>
</html>
```

כשר נרץ את הקוד בדף, נקבל את התוצאה הבאה:



2.8. טיפוס boolean |

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 (ומערכיהם) מייצגים את הערך true.

2.9. טיפוס null ו- undefined |

null הוא keyword language מייצג ערך מיוחד המשמש בדרך כלל לציין העדר ערך.

- הפעלת האופרטור typeof על null ממחזירה את המחרוזת object

undefined הוא הערך של כל משתנה שלא אוטחן, והוא מייצג גם את הערך המוחזר מפונקציות שאין להן ערך מוחזר.

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

- הפעלת האופרטור typeof על undefined ממחזירה את המחרוזת undefined

null ו- undefined מסמלים על היעדר ערך ויכולים לשמש לעיתים קרובות תחליף אחד לשני. אופרטור השווון == מחשב אותם שווים. (ואילו האופרטור === מחשב אותם שונים). אולם נפוץ להשתמש ב- undefined כדי ליזג היעדר ערך בرمת המערכת. וב- null למטרת איפוס אובייקטים שכבר אומחלו.

Wrapper Objects . 2.10 |

בכל פעם שמנסים לגש ל- property של מחרוזת, ממיר את ערך המחרוזת ל- object (כמו האובייקט שנקבל על ידי הפונקציה new String()).

האובייקט הזה ירש methods של string ומשמש בתור property reference. לאחר שהשימוש ב- property או ב- method הסתיים, האובייקט החדש שנוצר נמחק בצורה אוטומטית.

המספריים והבוליאנים משתמשים באותה שיטה: אובייקט זמני נוצר באמצעות הבנאי Number() או Boolean() ובאמצעות אובייקט זמני זה אפשר לגש ל- property או ל- method.

האובייקטים הזמניים שנוצרו בעת גישה למאפיין של מחרוזת, מספר או בוליאני ידועים כ- wrapper objects. ומאפייניהם הם לקרייה בלבד. לפיכך אם ננסה להגיד 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 לגש למאפיין של אחד מערכיהם אלה יגרום ל- TypeError.

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



String() , Number() , Boolean()

```
<!DOCTYPE html>
<html lang="en" xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta charset="utf-8" />
    <title></title>

<script>
    var s = "test";      //string - primitive
    var n = 1;           //number - primitive
    var b = true;        // boolean - primitive

    var S = new String(s);      //String object
    var N = new Number(n);      //Number object
```

```

var B = new Boolean(b);           //Boolean object

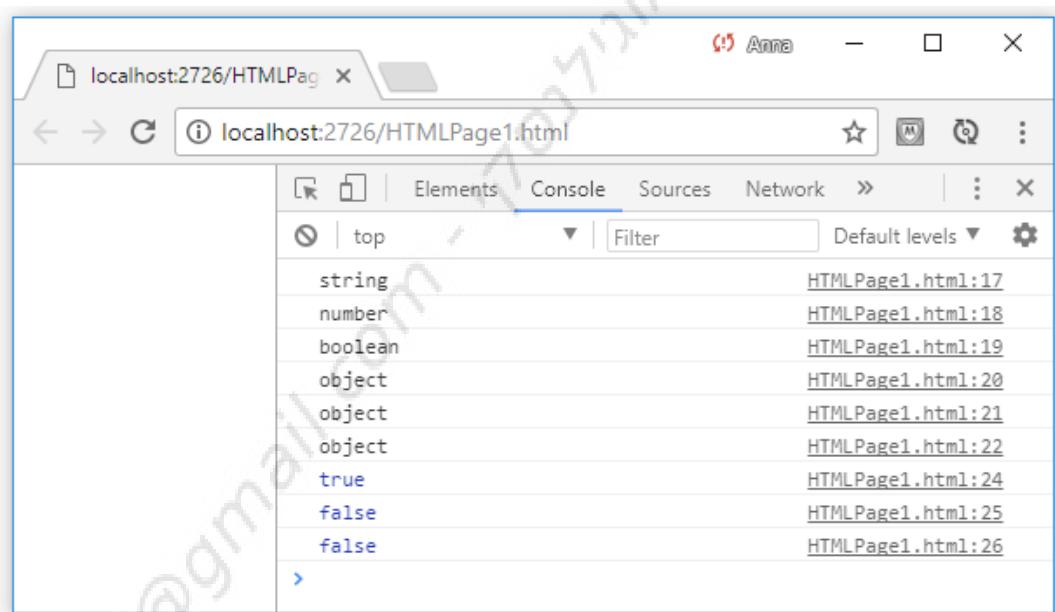
console.log(typeof (s));
console.log(typeof (n));
console.log(typeof (b));
console.log(typeof (S));
console.log(typeof (N));
console.log(typeof (B));

console.log(b==B);
console.log(b === B);
console.log(typeof (b)==typeof (B));
</script>
</head>
<body>

</body>
</html>

```

כasher נרץ את הקוד בדפדפן, נקבל את התוצאה הבאה:



האובייקטים N, S, ו- B בדוגמה לעיל יתנהגו בדרך כלל, בדיק כomo הערכים b, n, s, .wrapper object typeof === יראו את ההבדל בין primitive value ל' global object .2.11 |

global object .2.11 |

האובייקט global הוא אובייקט JavaScript רגיל המשרת מטרה חשובה מאוד: המאפיינים של אובייקט זה הם globally defined symbols הzmnimim לתוכנית JavaScript .
כasher ה- interpreter של JavaScript טוען דף חדש, הוא יוצר אובייקט גלובלי חדש ומעניק לו קבוצה ראשונית של מאפיינים המגדירים:

- מאפיינים גלובליים כמו NaN, undefined, Infinity
- פונקציות גלובליות כמו eval ו-isNaN
- 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 בחילון הדף שבו מיצג.

Object and array initializers .2.12

array initializer הוא רשימה מופרדת בפסיקים של ביטויים הכלולים בסוגרים 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 להיכל באיתחול 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 להגיד מערך בדרך הבאה:

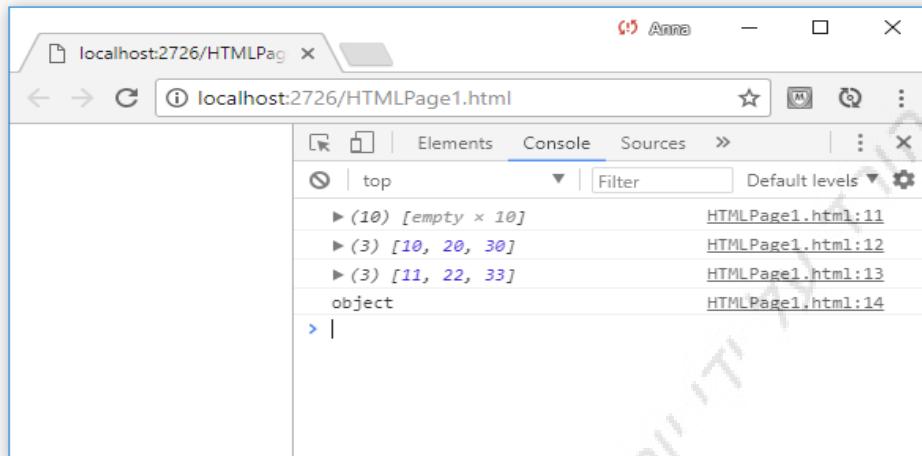
```
var arr = new Array(10)
```

להלן דוגמה מלאה:

```
<html lang="en" xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta charset="utf-8" />
    <title></title>
    <script>
        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);
    </script>
</head>
<body>

</body>
</html>
```

והתוצאה תהיה:



שים לב: ניתן להוסיף למערכות איברים בצורה דינמית.

Object initializer הוא כמו array initializer, אך הסוגרים המרובעים מוחלפים בסוגרים מסולסלים, וכל subexpression מורכב ממפתח המאפיין + נקודותים + ערך המאפיין. לדוגמה:

```

<!DOCTYPE html>

<html lang="en" xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta charset="utf-8" />
    <title></title>

    <script>
        // An object with 2 properties
        var p = { x: 2.3, y: -1.2 };

        //An empty object with no properties
        var q = {};

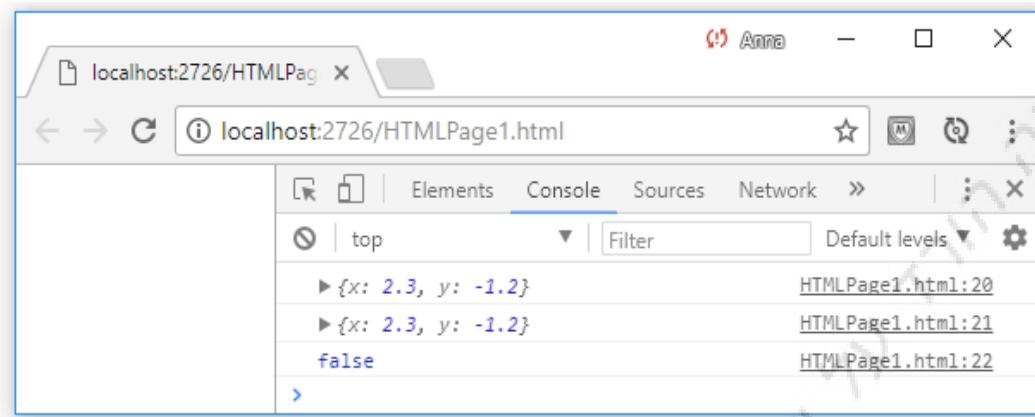
        //add to q the same properties as p
        q.x = 2.3;
        q.y = -1.2;

        console.log(p);
        console.log(q);
        console.log(p==q);
    </script>
</head>
<body>

</body>
</html>

```

© כל הזכויות שמורות לג'ון ברייס הדרכה בע"מ מקבוצת מטריקס



אפשרם להיות מוכנים. לדוגמה:

```

<!DOCTYPE html>

<html lang="en" xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta charset="utf-8" />
    <title></title>

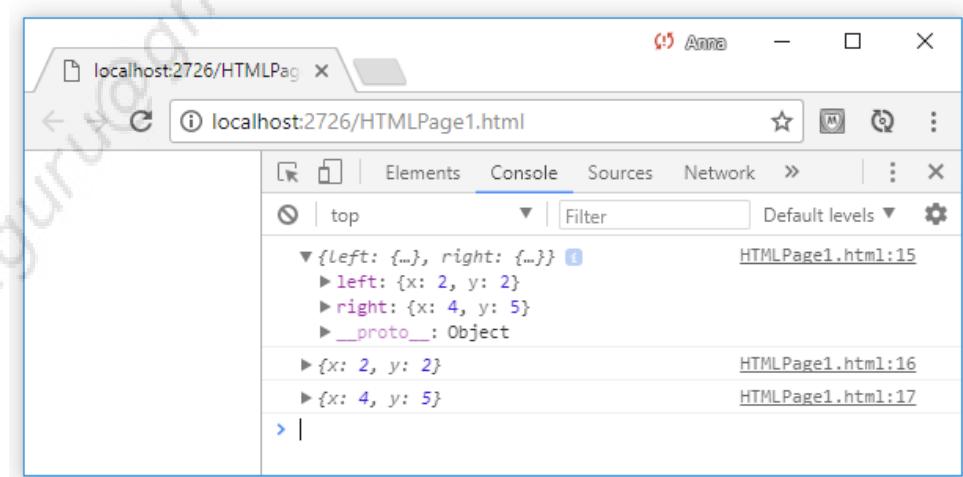
    <script>

        var linearLine = {
            left: { x: 2, y: 2 },
            right: { x: 4, y: 5 }
        };

        console.log(linearLine);
        console.log(linearLine.left);
        console.log(linearLine.right);

    </script>
</head>
<body>
</body>
</html>

```



שיםו לב: ניתן להוסיף לאובייקטים מאפיינים בצורה דינמית.



תרגיל



נתונה הפונקציה הבאה:

```
function test(x) {  
    return x != x;  
}
```

איזה ערך יכול להישלח לפונקציה זו, ב כדי שהוא תחזיר **true**?

3. הגדרת משתנים – Variable Declaration

VAR .3.1 |

מאז הייסודה, לא היה דרך אחות להכריז על משתנים: var. הצהרת משתנים באמצעות var, עובדת לפי עקרון ה- variable ופועלת כאילו המשתנים הוכרזו בראש ה execution context הנוכחי (פונקציה).

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

```
<!DOCTYPE html>

<html lang="en" xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta charset="utf-8" />
    <title></title>
    <script>
        function func() {

            // Intended to write to a global variable named 'num1'.
            num1 = 4;

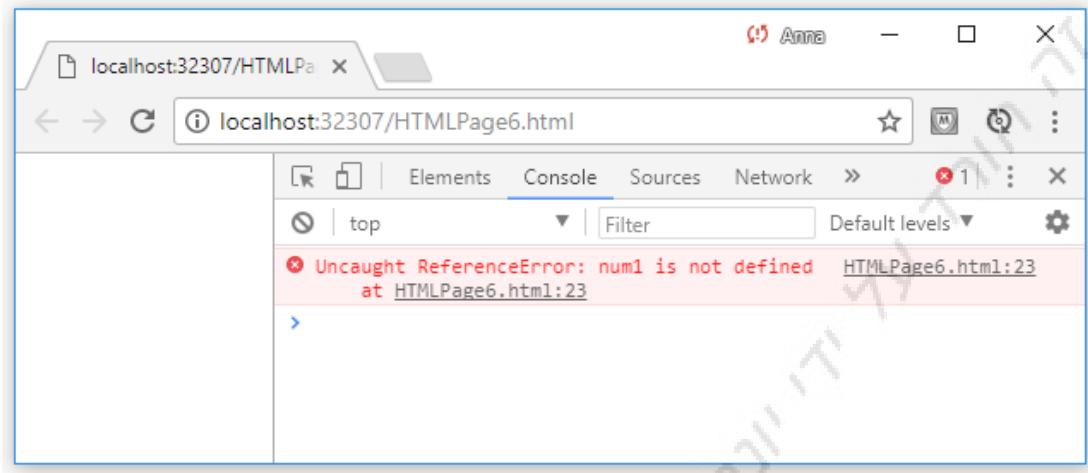
            if (num1 == 4) {
                // This declaration is moved to the top,
                //causing the first write to 'num1' to act on the local variable
                //rather than a global one.
                var num1 = 3;
            }
        }

        func();

        console.log(num1); //should print 4 but results in an exception.
    </script>
</head>
<body>

</body>
</html>
```

כאשר נריץ את הקוד בדף, נקבל את התוצאה הבאה:



עבור תוכניות גדולות - hoisting של משתנה יכול לגרום להתנחות בלתי צפואה וכן מציגה שתי דרכים חדשות להכרזה על משתנים: 2015

- `let`
- `const`

let .3.2 |

הכרת `let` פועלת בדיקן כמו הכרת `var`, אך עם הבדל גדול: ההצהרות מוכנות רק בבלוק המופיע את המשתנה, זמינים רק מהנקודה שבה ההצהרה מזוקמת. לכן המשתנים המוצזרים על ידי `let` בתוך לולה, או פשוט בתוך סוגרים מסווגים, תקפים רק בתחום הבלוק הזה, ורק לאחר מכן תן הכרה. התנחות 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 בлок פנימי של הפונקציה) – למעשה, אנחנו יכולים להזכיר פעמיים משתנה `var`, אבל זה לא ייצור משתנה חדש, אלא עדין יתיחס למשתנה בעל השם הזה שהוגדר קודם לכן באותו הפונקציה.
- `let` – ניתן ליצור משתנה בעל שם זהה לבlok החיצוני בתוך בлок פנימי הגדרה זו תיצור משתנה חדש שיטיל כל על המשתנה החיצוני (אפקט ה- `shadow`)

הנה כמה דוגמאות שיפשו את הקונספט הנ"ל:

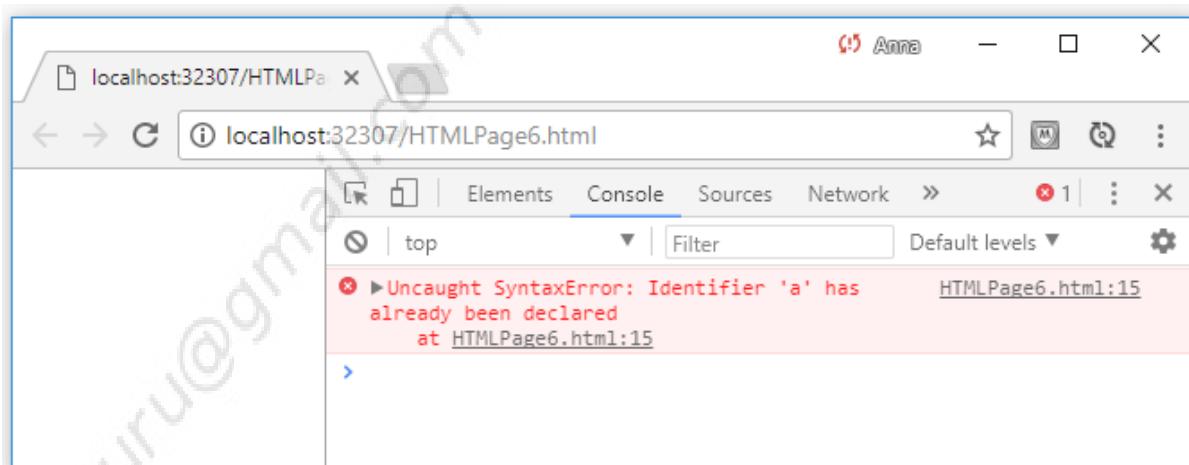
דוגמה ראשונה:

```
<!DOCTYPE html>

<html lang="en" xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta charset="utf-8" />
    <title></title>
    <script>
        function setLetAndVar(){
            let a=4;
            {
                var a;
                console.log(a);
            }
            setLetAndVar();
        </script>
    </head>
    <body>

    </body>
</html>
```

כאשר נרץ את הקוד בדף, נקבל את התוצאה הבאה:



דוגמיה שנייה:

```
<!DOCTYPE html>

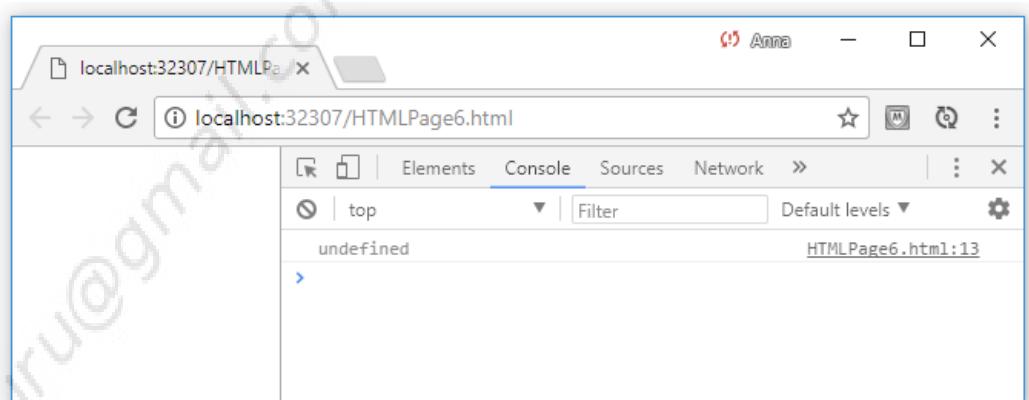
<html lang="en" xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta charset="utf-8" />
    <title></title>
    <script>

        function setLetAndVar() {
            let a = 4;
            {
                let a;
                console.log(a); //output: 4
            }
        }

        setLetAndVar();
    </script>
</head>
<body>

</body>
</html>
```

כasher נרץ את הקוד בדף, נקבל את התוצאה הבאה:



דוגמה שלישית:

```
<!DOCTYPE html>

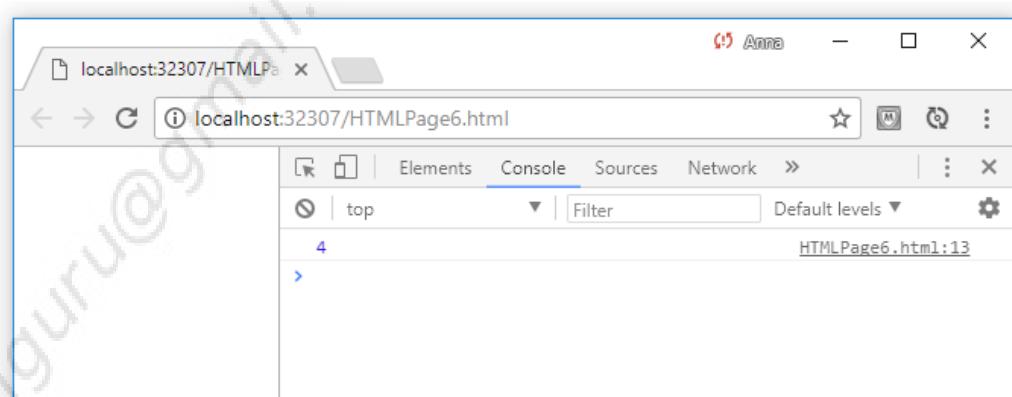
<html lang="en" xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta charset="utf-8" />
    <title></title>
    <script>

        function setLetAndVar() {
            var a = 4;
            {
                var a;
                console.log(a); //output: 4
            }
        }

        setLetAndVar();
    </script>
</head>
<body>

</body>
</html>
```

כאשר נרץ את הקוד בדף, נקבל את התוצאה הבאה:



דוגמאות רביעית:

```
<!DOCTYPE html>

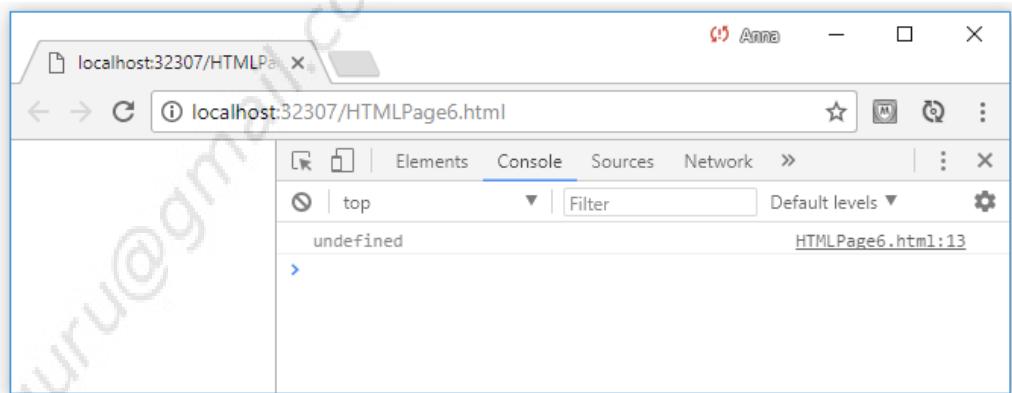
<html lang="en" xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta charset="utf-8" />
    <title></title>
    <script>

        function setLetAndVar() {
            var a = 4;
            {
                let a;
                console.log(a); //output: undefined
            }
        }

        setLetAndVar();
    </script>
</head>
<body>

</body>
</html>
```

כasher נרץ את הקוד בדף, נקבל את התוצאה הבאה:



דוגמה חמישית:

```
<!DOCTYPE html>

<html lang="en" xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta charset="utf-8" />
    <title></title>
    <script>

        function setVar(){
            var b = 10;

            console.log("var - outer block (step 1): ", b);

            {
                console.log("var - inner block (step 2): ", b);

                var b = 11;      //will change the value of the function's scope var b

                console.log("var - inner block (step 3): ", b);
            }

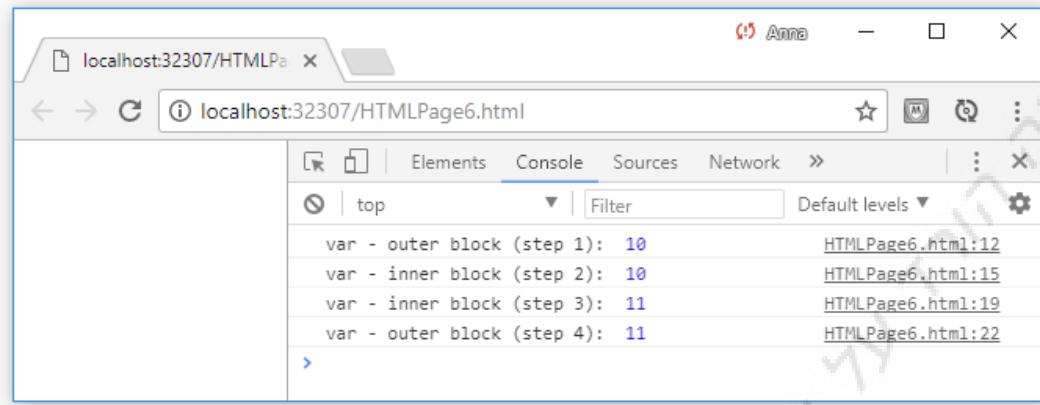
            console.log("var - outer block (step 4): ", b);
        }

        setVar();

    </script>
</head>
<body>

</body>
</html>
```

כasher נרץ את הקוד בדף, נקבל את התוצאה הבאה:



דוגמאות שישית:

```
<!DOCTYPE html>

<html lang="en" xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta charset="utf-8" />
    <title></title>
    <script>
        function setLet() {
            let a = 10;

            console.log("let - outer block (step 1): ", a);
            {
                console.log("let - inner block (step 2): ", a);

                let a = 11;

                console.log("let - inner block (step 3): ", a);
            }

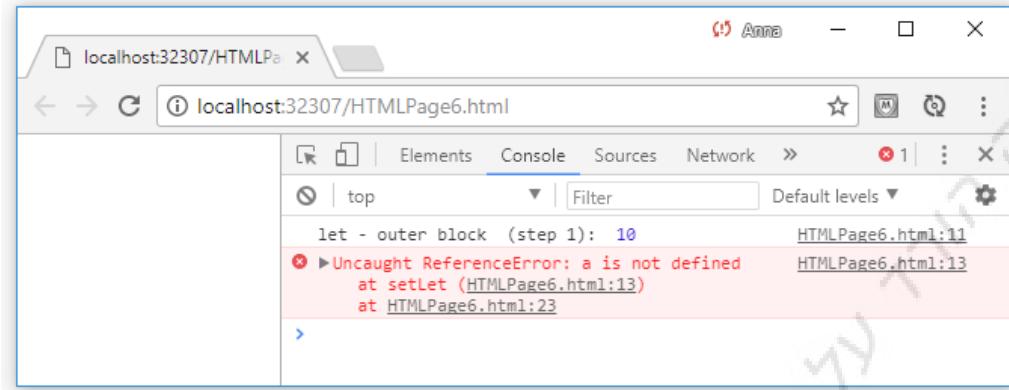
            console.log("let - outer block (step 4): ", a);
        }

        setLet();

    </script>
</head>
<body>

</body>
</html>
```

כאשר נרץ את הקוד בדף, נקבל את התוצאה הבאה:



דוגמה שביעית:

```
<!DOCTYPE html>

<html lang="en" xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta charset="utf-8" />
    <title></title>
    <script>
        function setLet() {
            let a = 10;

            console.log("let - outer block (step 1): ", a);
            {
                //console.log("let - inner block (step 2): ", a);

                let a = 11;

                console.log("let - inner block (step 3): ", a);
            }

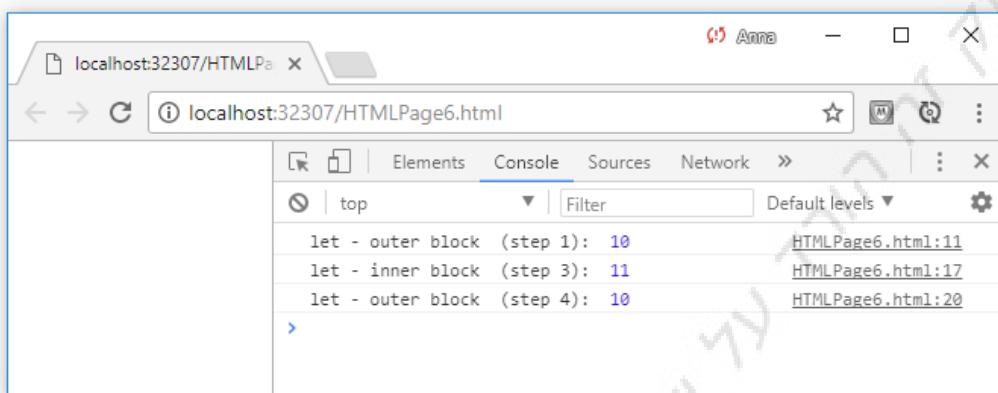
            console.log("let - outer block (step 4): ", a);
        }

        setLet();

    </script>
</head>
<body>

</body>
</html>
```

כasher נרץ את הקוד בדף, נקבל את התוצאה הבאה:



Const .3.3 |

הצורה משתנה קבוע ב JavaScript מוצעת באמצעות המילה `const`.
כל הגדרת משתנה על ידי `const` מחייבת להשים ערך לתוך המשתנה בשורה בה הוא מוגדר.
לדוגמה:

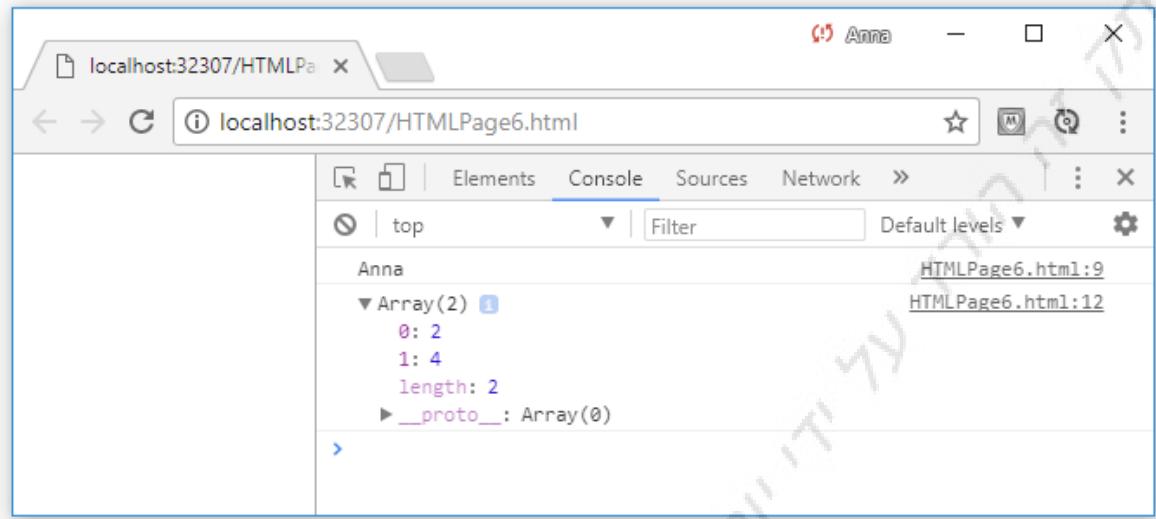
```
<!DOCTYPE html>

<html lang="en" xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta charset="utf-8" />
    <title></title>
    <script>
        var obj = "Anna"; //obj is bound to the primitive string "Anna".
        console.log(obj);

        obj = [2, 4]; // obj is now bound to an array object.
        console.log(obj);
    </script>
</head>
<body>

</body>
</html>
```

כasher נרץ את הקוד בדף, נקבל את התוצאה הבאה:



הצורת ה - `const`, בניגוד להצורות ה - `let` ו- `var`, אינה מאפשרת לשנות את המשתנה לאחר ההצירה הראשונית:

```

<!DOCTYPE html>

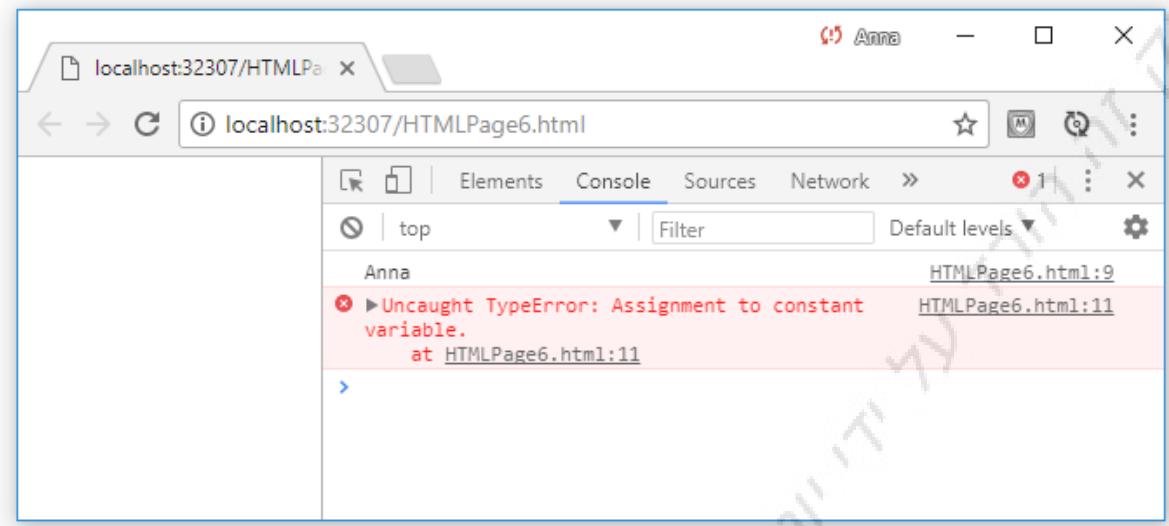
<html lang="en" xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta charset="utf-8" />
    <title></title>
    <script>
        const obj = "Anna"; //obj is bound to the primitive string "Anna".
        console.log(obj);

        obj = [2, 4]; // TypeError
        console.log(obj);
    </script>
</head>
<body>

</body>
</html>

```

כasher נרץ את הקוד בדף, נקבל את התוצאה הבאה:



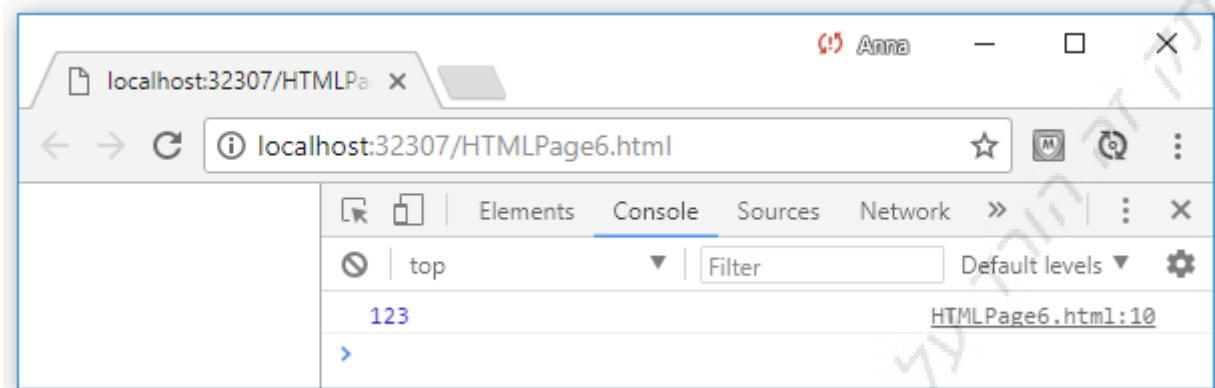
אינו הופך את הערך ל immutable const

פירושו של משתנה יש תמיד אותו ערך, אך אין פירוש הדבר שהערך עצמו הוא הופך להיות **immutable** (בלתי משתנה).
לדוגמא, `obj` הוא קבוע, אך הערך שהוא מצביע עליו הוא **mutable** (ניתן לשינוי) - ואני יכולם להוסיף לו **property** (מאפיין):

```
<!DOCTYPE html>

<html lang="en" xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta charset="utf-8" />
    <title></title>
    <script>
        const obj = {};
        obj.x = 123;
        console.log(obj.x);
    </script>
</head>
<body>
</body>
</html>
```

כasher נרץ את הקוד בדף, נקבל את התוצאה הבאה:



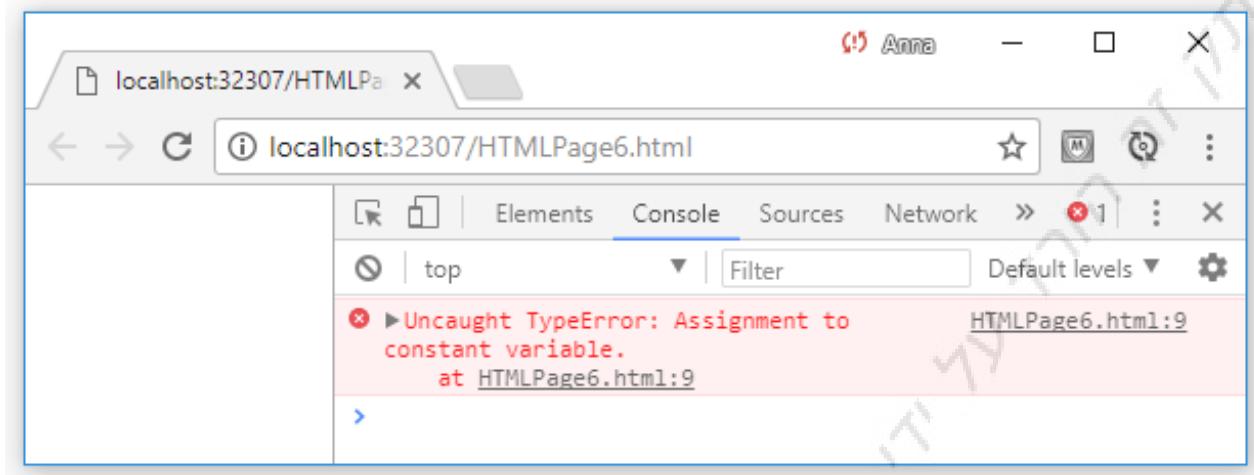
עם זאת, איננו יכולים לבצע השמה של ערך אחר לתוך המשתנה obj :

```
<!DOCTYPE html>

<html lang="en" xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta charset="utf-8" />
    <title></title>
    <script>
        const obj = {};
        obj = 123;
        console.log(obj.x);
    </script>
</head>
<body>

</body>
</html>
```

כאשר נריץ את הקוד בדף, נקבל את התוצאה הבאה:



Temporal dead zone .3.4 |

למשתנה שהוכרז על ידי `let` או `Const` יש איזור מת זמני (TDZ): כאשר הקוד מגיע לבLOCK המקיים של אותו משתנה (scope), לא ניתן לגשת אליו (לקבל את תוכן המשתנה או להציג ערך למשתנה) עד שתבוצע השורה של הוצאה.

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. מחזורי החיים שלהם כוללים את השלבים הבאים:
- כאשר הביצוע של הקוד מגיע ל `scope` של המשתנה(הפונקציה בה המשתנה מוגדר), מוקצת מידית שטח אחסון (כולל binding) עבורו, והמשתנה מאוחול מיד, על ידי הערך `undefined`.
 - כאשר הביצוע של הקוד בתוך ה `scope` מגיע להוצאה, המשתנה מקבל את הערך שצויין על ידי האתחול (assignment) - אם קיים.
אם אין `initializer`, הערך של המשתנה נשאר `undefined`.

The life cycle of let-declared variables

- משתנים שהוכרזו על ידי `let` מכילים מחזורי החיים שלהם הוא המחזור הבא:
- כאשר הביצוע של הקוד מגיע ל `scope` (הBLOCK המקיים אותו) של משתנה `let`, נוצר שטח אחסון (כולל binding) עבורו. המשתנה נשאר `uninitialized`.
 - גישה למשתנה במצב `uninitialized` גורמת ל `ReferenceError`.

- כאשר הביצוע של הקוד בתוך ה- scope מגיע להצורה, 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 למשתני let, אבל הם חייכים להיות מתחלים בשורת ההגדירה (כלומר, לקבל ערך מייד בשורת ההגדירה) ולא ניתן לשנות אותם בהמשך ה-scope.

דוגמאות:

בתוך TDZ יזרק exception אם ננסה לבצע למשתנה פעולות get / set:

```
<!DOCTYPE html>

<html lang="en" xmlns="http://www.w3.org/1999/xhtml">
<head>
  <meta charset="utf-8" />
  <title></title>
  <script>

    let tmp = true;
    if (true) { // enter new scope, TDZ starts - Uninitialized binding for `tmp` is created

      console.log(tmp); // ReferenceError

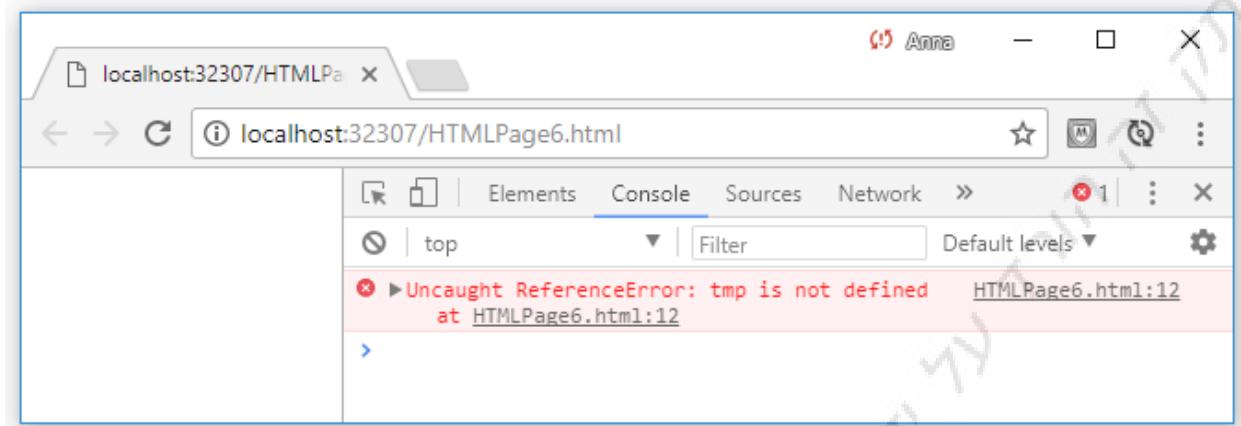
      let tmp; // TDZ ends, `tmp` is initialized with `undefined`
      console.log(tmp); // undefined

      tmp = 123;
      console.log(tmp); // 123
    }
    console.log(tmp); // true

  </script>
</head>
<body>

</body>
</html>
```

כasher נרץ את הקוד בדפדפן, נקבל את התוצאה הבאה:



Serving from the file system? Add your files into the workspace. more never show

```

1 <!DOCTYPE html>
2
3 <html lang="en" xmlns="http://www.w3.org/1999/xhtml">
4 <head>
5   <meta charset="utf-8" />
6   <title></title>
7   <script>
8
9     let tmp = true;
10    if (true) { // enter new scope, TDZ starts - Uninitialized binding for `tmp` is created
11
12      console.log(tmp); // ReferenceError
13
14      let tmp; // TDZ ends, `tmp` is initialized with `undefined`
15      console.log(tmp); // undefined
16
17      tmp = 123;
18      console.log(tmp); // 123
19    }
20    console.log(tmp); // true
21
22  </script>
23 </head>
24 <body>
25
26 </body>
27 </html>
```

A callout bubble points to the line `console.log(tmp); // ReferenceError` with the text `Uncaught ReferenceError: tmp is not defined`.

אם יש אתחול AZ TDZ מותרים לאחר ביצוע ה- initializer

```

<!DOCTYPE html>

<html lang="en" xmlns="http://www.w3.org/1999/xhtml">
<head>
  <meta charset="utf-8" />
  <title></title>
  <script>
```

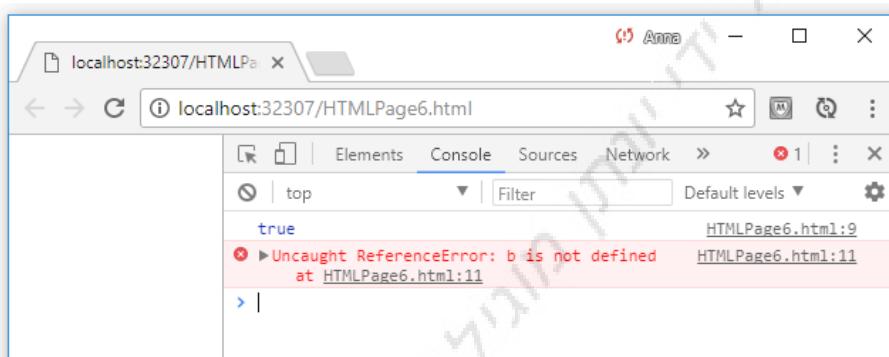
```

var a = !a; //ok
console.log(a);

let b = !b; // ReferenceError
console.log(b);
</script>
</head>
<body>
</body>
</html>

```

כשר נרץ את הקוד בדף, נקבל את התוצאה הבאה:



הקוד הבא מדגים כי ה - **dead zone** הוא באמת זמן (על פי הזמן) ולא מרחב (על פי מקום):

```

<!DOCTYPE html>

<html lang="en" xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta charset="utf-8" />
    <title></title>
    <script>
        { // enter new scope, TDZ starts

            const func = function () {
                console.log(myVar); // OK!
            };

            // Here we are within the TDZ and
            console.log(myVar); // ReferenceError

            let myVar = 3; // TDZ ends
            func(); // called outside TDZ
        }

    </script>
</head>
<body>

</body>

```

</html>

כasher נורץ את הקוד בדף, נקבל את התוצאה הבאה:

```

1 <!DOCTYPE html>
2
3 <html lang="en" xmlns="http://www.w3.org/1999/xhtml">
4 <head>
5   <meta charset="utf-8" />
6   <title></title>
7   <script>
8     { // enter new scope, TDZ starts
9
10       const func = function () {
11         console.log(myVar); // OK!
12       };
13
14       // Here we are within the TDZ and
15       console.log(myVar); // ReferenceError
16
17       let myVar = 3; // TDZ ends
18       func(); // called outside TDZ
19     }
20
21   </script>
22 </head>
23 <body>
24
25 </body>
26 </html>
27

```

```

<!DOCTYPE html>

<html lang="en" xmlns="http://www.w3.org/1999/xhtml">
<head>
  <meta charset="utf-8" />
  <title></title>
  <script>
    { // enter new scope, TDZ starts

      const func = function () {
        console.log(myVar); // OK!
      };

      // Here we are within the TDZ and
      //console.log(myVar); // ReferenceError

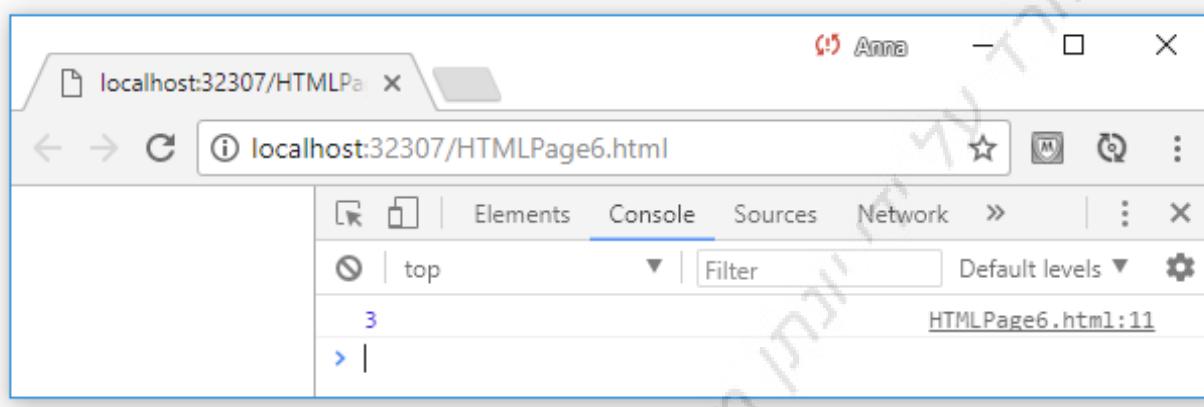
      let myVar = 3; // TDZ ends
      func(); // called outside TDZ
    }

    </script>
</head>

```

```
<body>
</body>
</html>
```

כasher נרץ את הקוד בדף, נקבל את התוצאה הבאה:



ערכי ברירת המחדל של פרמטרים | temporal dead zone

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

```
<!DOCTYPE html>

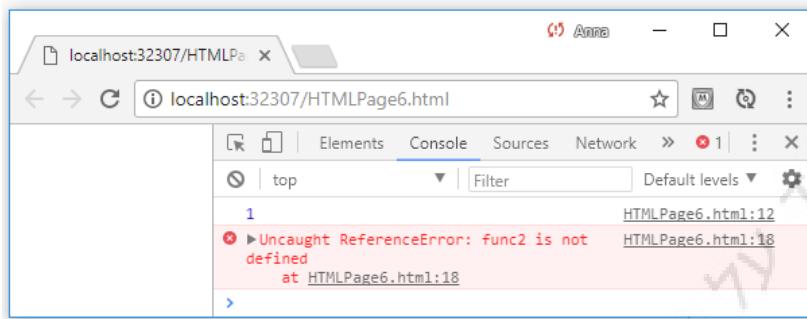
<html lang="en" xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta charset="utf-8" />
    <title></title>
    <script>
        // OK: `y` accesses `x` after it has been declared
        function func1(x = 1, y = x) {
            return y;
        }
        console.log(func1()); // 1

        // Exception: `x` tries to access `y` within TDZ
        function bar(x = y, y = 1) {
            return x;
        }
        console.log(func2()); // ReferenceError

    </script>
</head>
<body>

</body>
</html>
```

כאשר נריץ את הקוד בדף, נקבל את התוצאה הבאה:



3.5. לוֹלָאוֹת וְהַגְּדָרָת מִשְׁתְּנִים |

להלן הלוֹלָאוֹת הבאות מאפשרות לך להכיר על משתנים בראשם:

- for
- for-in
- for-of

כדי להציג על המשתנים, אפשר להשתמש בו- `let`, `var` או `const`.
כאשר לכל אחד מהם יש השפעה שונה.

for loop

`var` - הכרזה על משתנה בראש לוֹלָאָה על ידי `var` יוצרת storage space עבור משתנה זה:

```
<!DOCTYPE html>

<html lang="en" xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta charset="utf-8" />
    <title></title>
    <script>
        const arr = [];
        for (var i = 0; i < 3; i++) {
            arr.push(() => i);
        }
        arr.map(x => console.log(x()));
    </script>
</head>
<body>
</body>
</html>
```

© כל הזכויות שמורות לגיאן בריס הדרכה בע"מ מקבוצת מטריקס

כasher nariz at ha-kod b-dafon, nikbel at ha-totzaa ha-bava:

	Filter	Default levels ▾	⚙
3		HTMLPage1.html:12	
3		HTMLPage1.html:12	
3		HTMLPage1.html:12	

כל זה בגופם של שלושה arrow functions מתייחסים לאותו binding, ולכן יכולים ייחזרו את אותו ערך.

אולם, אם הגדרת המשתנה תבוצע על ידי let, ייווצר binding חדש עבור כל איטרציה בולולאה:

```
<!DOCTYPE html>

<html lang="en" xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta charset="utf-8" />
    <title></title>
    <script>
        const arr = [];
        for (let i = 0; i < 3; i++) {
            arr.push(() => i);
        }
        arr.map(x => console.log(x()));
    </script>
</head>
<body>
</body>
</html>
```

כasher nariz at ha-kod b-dafon, nikbel at ha-totzaa ha-bava:

localhost:61220/HTMLPage1.html	
Console	Elements
top	Default levels ▾
0	HTMLPage1.html:12
1	HTMLPage1.html:12
2	HTMLPage1.html:12

הפעם, כל אחד מתייחס ל binding של איטרציה מסוימת ומשמר את הערך שהוא קיים באותו זמן. לכן, כל const arrow function מחזיר ערך שונה.

עובד כמו var אבל אתה אי אפשר לשנות את הערך המקורי של המשתנה:

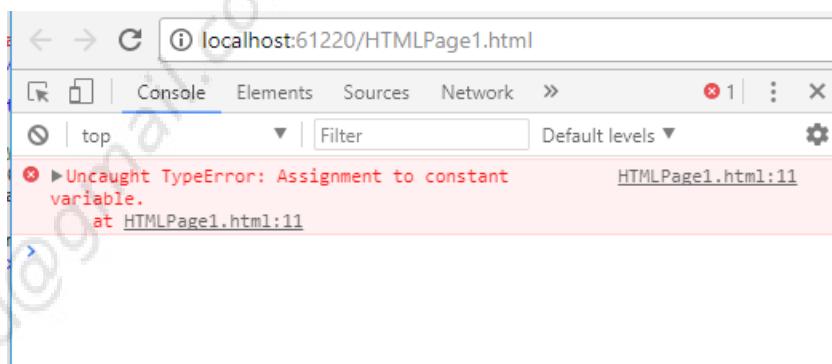
```
<!DOCTYPE html>

<html lang="en" xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta charset="utf-8" />
    <title></title>
    <script>
        const arr = [];

        // TypeError: Assignment to constant variable due to i++
        for (const i = 0; i < 3; i++) {
            arr.push(() => i);
        }
        arr.map(x => console.log(x()));
    </script>

</head>
<body>
</body>
</html>
```

כasher נרץ את הקוד בדפדפן, נקבל את התוצאה הבאה:



for-of loop and for-in loop

ב - loop, הגדרת משתנה על ידי var תיצור binding יחיד:

```
<!DOCTYPE html>

<html lang="en" xmlns="http://www.w3.org/1999/xhtml">
<head>
  <meta charset="utf-8" />
  <title></title>
  <script>
    const arr = [];

    for (var i of[0, 1, 2]) {
      arr.push(() => i);
    }

    arr.map(x => console.log(x()));
  </script>

</head>
<body>
</body>
</html>
```

כasher נרץ את הקוד בדף, נקבל את התוצאה הבאה:

▼ Filter	Default levels ▼	⚙
3	HTMLPage1.html:12	
3	HTMLPage1.html:12	
3	HTMLPage1.html:12	

ויצור immutable binding Const אחד לכל איטרציה:

```
<!DOCTYPE html>

<html lang="en" xmlns="http://www.w3.org/1999/xhtml">
<head>
  <meta charset="utf-8" />
  <title></title>
  <script>
    const arr = [];

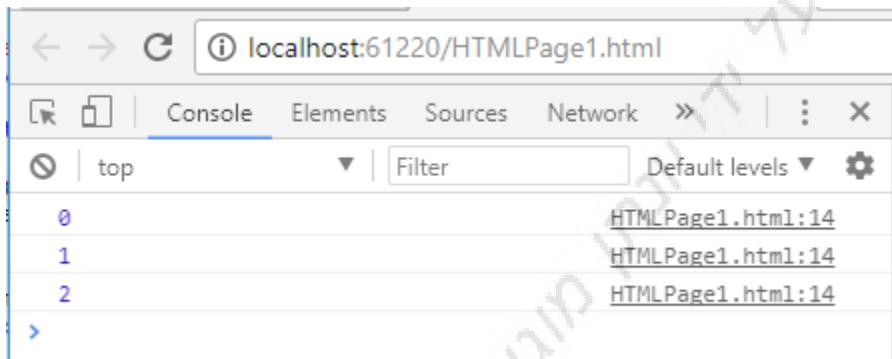
    for (const i of[0, 1, 2]) {
      arr.push(() => i);
    }

    arr.map(x => console.log(x()));
  </script>
```

```
</head>
<body>
</body>

</html>
```

כasher נרץ את הקוד בדפדפן, נקבל את התוצאה הבאה:



mutable אחד לכל איטרציה, אבל ה **binding** שהוא יוצר מוגדר כ- **let**

```
<!DOCTYPE html>

<html lang="en" xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta charset="utf-8" />
    <title></title>
    <script>
        const arr = [];

        for (let i of[0, 1, 2]) {
            arr.push(() => i);
        }

        arr.map(x => console.log(x()));
    </script>

</head>
<body>
</body>

</html>
```

כאשר נריץ את הקוד בדפדפן, נקבל את התוצאה הבאה:

```
localhost:61220/HTMLPage1.html
Console Elements Sources Network ...
top Filter Default levels ...
0 HTMLPage1.html:14
1 HTMLPage1.html:14
2 HTMLPage1.html:14
```

לולאת `for-in` פועלת באופן דומה לlolאת `of-for`

3.6. סיכום אופני הגדרת משתנים

הטבלה הבאה מציגה סקירה של הדרכים בהן ניתן להגדיר משתנים ב-ES6:

	Hoisting	Scope	Creates global properties
<code>var</code>	Declaration	Function	Yes
<code>let</code>	Temporal dead zone	Block	No
<code>const</code>	Temporal dead zone	Block	No
<code>function</code>	Complete	Block	Yes

4. פונקציות

ב- JavaScript, פונקציות הן אובייקטים, וניתן להקנות פונקציות לתוך משתנים ולהעביר אותם לפונקציות אחרות.

הגדרות פונקציית JavaScript יכולות להיות מקוונות בפונקציות אחרות, ויש להן גישה לכל המשתנים הנמצאים ב שום scope שלהם, כאשר הם מוגדרים. שימושות הדבר היא כי פונקציות Java Script הם closures, וזהו טכניקה חשובה, אותה נסקור בפירוט בהמשך הפרק.

4.1. דרכי להגדרת פונקציות

- functions as statement

```
function f(x) {  
    return 1;  
};
```

- functions as expressions

שם הפונקציה הוא אופציוני עברו פונקציות המוגדרות כביטויים. ביטוי הצהרת פונקציה למשה מכיר על משתנה ומקרה לתוכו אובייקט פונקציה.
ביטוי להגדרת פונקציה:

```
var f = function (x) { return 1; };
```

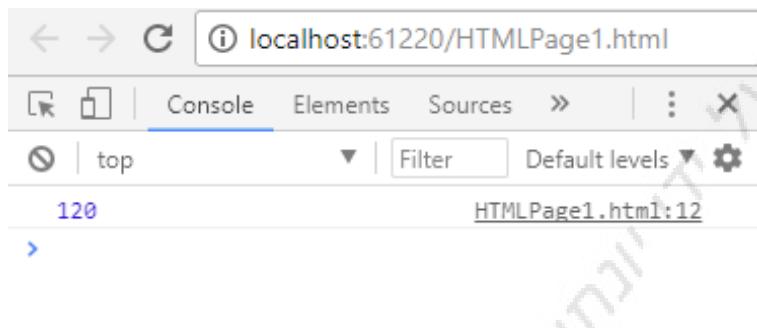
רוב הפונקציות המוגדרות כביטויים אינם זוקקים לשם פונקציה, מה שהופך את ההגדירה שלהם קומפקטיבית יותר.

בכל מקרה, גם לפונקציות המוגדרות כביטויים מותר לתת שם, כמו בדוגמה הבאה:

```
<!DOCTYPE html>  
  
<html lang="en" xmlns="http://www.w3.org/1999/xhtml">  
<head>  
    <meta charset="utf-8" />  
    <title></title>  
    <script>  
        var f = function fact(x) {  
            if (x <= 1) return 1; else return x * fact(x - 1);  
        };  
  
        console.log(f(5))
```

```
</script>
</head>
<body>
</body>
</html>
```

כasher neriz at haqod b'dafon, nikbel at ha'tozaza ha'babah:



דוגמה זו צריכה להתייחס לעצמה. וכן - אם ביטוי בהגדרת פונקציה כולל שם, היקף הפונקציה המקומית (local function scope) עבור פונקציה זו יכול binding אל השם של אובייקט הפונקציה. למעשה, שם הפונקציה הופך למשנה מקומי בתוך הפונקציה.

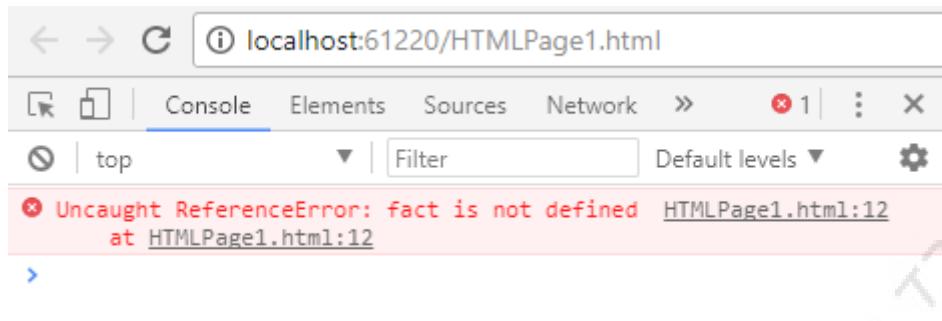
אבל אם ננסה לקרוא לפונקציה fact() באמצעות השם של הפונקציה, מוחוץ לקוד הבלוק של הפונקציה (), נקבל שגיאה, כפי שניתן לראות בדוגמה הבאה:

```
<!DOCTYPE html>

<html lang="en" xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta charset="utf-8" />
    <title></title>
    <script>
        var f = function fact(x) {
            if (x <= 1) return 1; else return x * f(x - 1);
        };

        console.log(fact(5))
    </script>
</head>
<body>
</body>
</html>
```

כasher neriz at haqod b'dafon, nikbel at ha'tozaza ha'babah:



function hoisting .4.2 |

הצהרות הצהרת פונקציה "מעולות" לחلكعلוין של ה- script המקיים או לחלקעלוין של הפונקציה המKİפה אוטם, כך שהפונקציות המצוחרות בדרך זו עשויות להיות מופעלות מקודם למופיע לפני שהן מוגדרות.

כל זה לא נכון עבור פונקציות המוגדרות כביטויים: כדי להפעיל פונקציה, חייבים להיות מסוגל להתייחס אליה, ואי אפשר להתייחס לפונקציה המוגדרת כביטוי עד ששורת ההגדלה של המשתנה בו היא מוקצת מתבצעת.

הערה: Variable declarations על ידי var מתבצעים בצורה hoisting בה המשתנים מוכרים כבר מראש השם, אך ההשمات של הערכים למשתנים אלו אינן מעולות מעלה, ולכן לא ניתן להפעיל פונקציות שהוגדרו עם ביטויים לפני שהקוד ביצע את שורת ההגדלה.



Nested functions .4.3 |

ב- JavaScript, פונקציות יכולות להיות מקוונות בתוך פונקציות אחרות. פונקציות מקוונות יכולות לחשוף פרמטרים ולמשתנים של הפונקציה (או הפונקציות) שהם מקוונים בתוכם:

```
<!DOCTYPE html>
<html lang="en" xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta charset="utf-8" />
    <title></title>
    <script>
        function f(a, b) {
            function square(x) { return x * x; }
```

© כל הזכויות שמורות לגיאן בריס הדרכה בע"מ מקבוצת מטריקס

```

        return Math.sqrt(square(a) + square(b));

    }

</script>

</head>
<body>
</body>
</html>

```

בקוד לעיל, הפונקציה הפנימית מרובע `square(s)` יכולה למסור אל הפרמטרים `a` ו- `b` שהוגדרו על ידי הפונקציה החיצונית `f()`.

Function overloading .4.4 |

JavaScript, לא תומכת בצורה דיפולטיבית בהעמסת פונקציות, אולם ישנו דרך לבצע "העמסה" מודומה. כדי שניתן לראות בדוגמה הבאה:

```

<!DOCTYPE html>

<html lang="en" xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta charset="utf-8" />
    <title></title>
    <script>

        // First way:
        function showMessage(message, header, footer) {

            var message2Show = "";

            if (header != undefined) {
                message2Show += header + "\n";
            }

            message2Show += message;

            if (footer != undefined) {
                message2Show += "\n" + footer;
            }

            console.log(message2Show);
        }
        showMessage("The exam will be next week.", "Note: ", "Good Luck");
        showMessage("The exam will be next week.", "Note: ");
        showMessage("The exam will be next week.");
    </script>

```

```
// Second way:
function showMessage(message, options) {

    var message2Show = "";

    if (options != undefined && options.header != undefined) {
        message2Show += options.header + "\n";
    }

    message2Show += message;

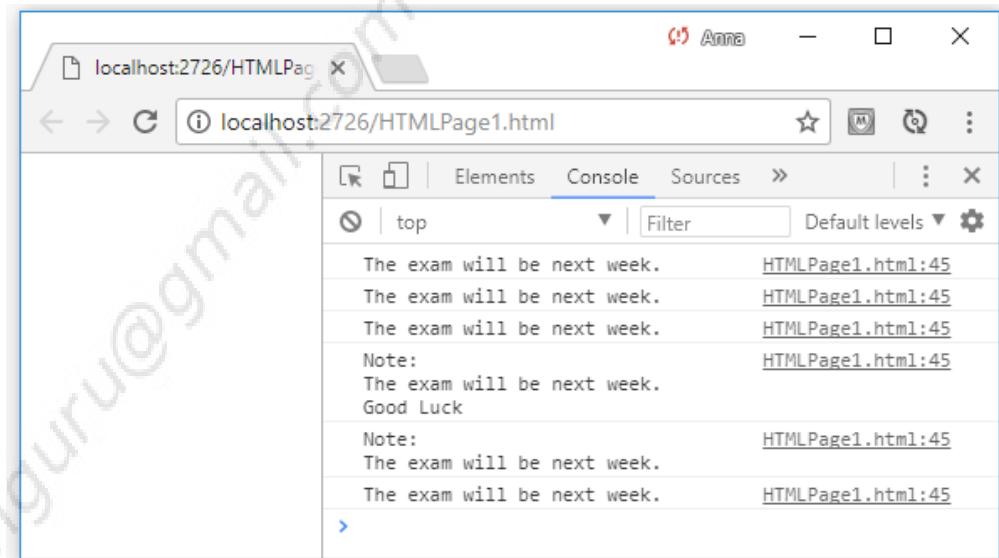
    if (options != undefined && options.footer != undefined) {
        message2Show += "\n" + options.footer;
    }

    console.log(message2Show);
}

showMessage("The exam will be next week.", { header: "Note: ", footer: "Good Luck" });
showMessage("The exam will be next week.", { header: "Note: " });
showMessage("The exam will be next week.");

</script>
</head>
<body>

</body>
</html>
```



Arrow functions .4.5 |

על אף העובדה multi-paradigm language JavaScript, עשויה שימוש בתוכנות פונקציונליות רבות. חלק מהתוכנות הללו הן closures ופונקציות אונונימיות.

ב-ES2015 נוספה לתוכנות האלו תכונה חדשה - arrow functions המאפשרת תחביר קצר יותר:

```
<!DOCTYPE html>

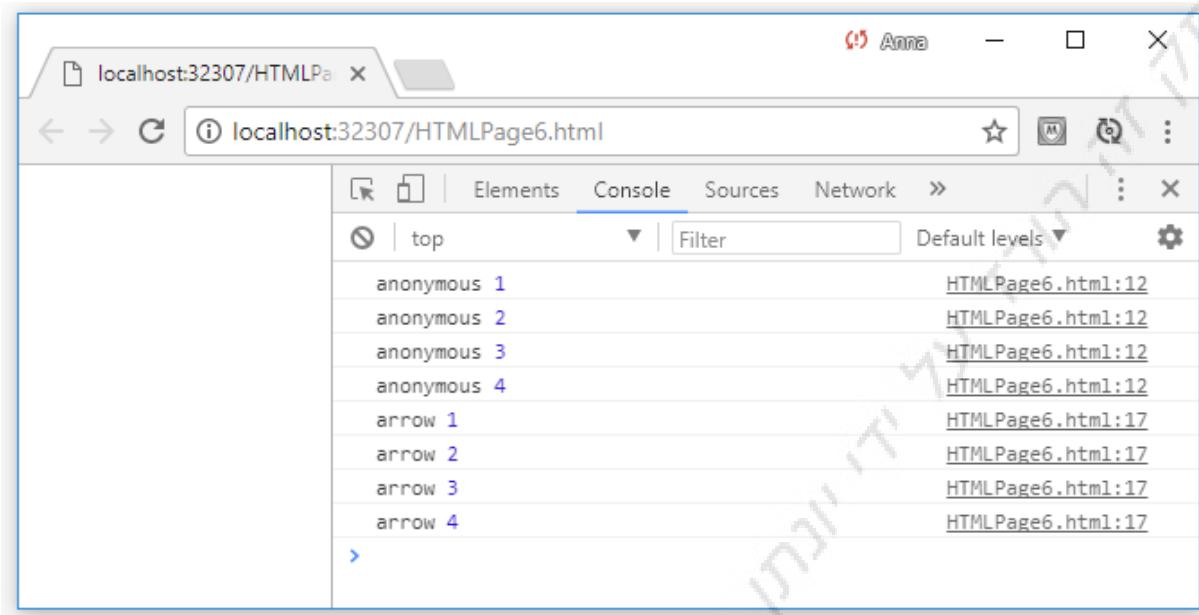
<html lang="en" xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta charset="utf-8" />
    <title></title>
    <script>
        var arr = [1, 2, 3, 4];

        // Before ES2015: anonymous function
        arr.forEach(function (element, index) {
            console.log("anonymous", element);
        });

        // After ES2015: arrow function
        arr.forEach((element, index) => {
            console.log("arrow", element);
        });

    </script>
</head>
<body>
</body>
</html>
```

כasher נרץ את הקוד בדפדפן, נקבל את התוצאה הבאה:



בהתחלת זה אולי נראה כמו שיפור קטן. עם זאת, arrow functions מתנהגות בצורה שונה מאשר כה שמדובר ב-`this`.

arrow functions ירושות את הערך `this` מהפונקציה המקיפה אותם. בנגדוד לפונקציות רגילות שלא ירושות את הערך `this` מהפונקציה המקיפה אותם, כפי שניתן לראות בדוגמה הבאה:

```
<!DOCTYPE html>

<html lang="en" xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta charset="utf-8" />
    <title></title>
    <script>
        function Calc() {
            this.num = 100;

            console.log(this.num);

            setTimeout(function callback() {

                // "this" points to the global object (or undefined - in strict mode)
                console.log(this.num);

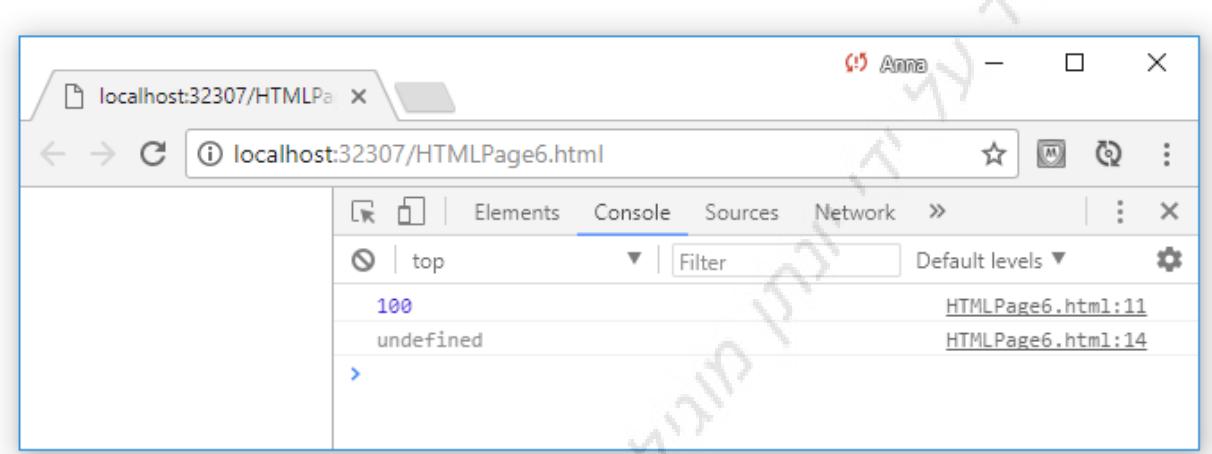
            }, 1000);
        }

        var calc = new Calc();

    </script>
</head>
<body>
```

```
</body>  
</html>
```

כאשר נריץ את הקוד בדף, נקבל את התוצאה הבאה:

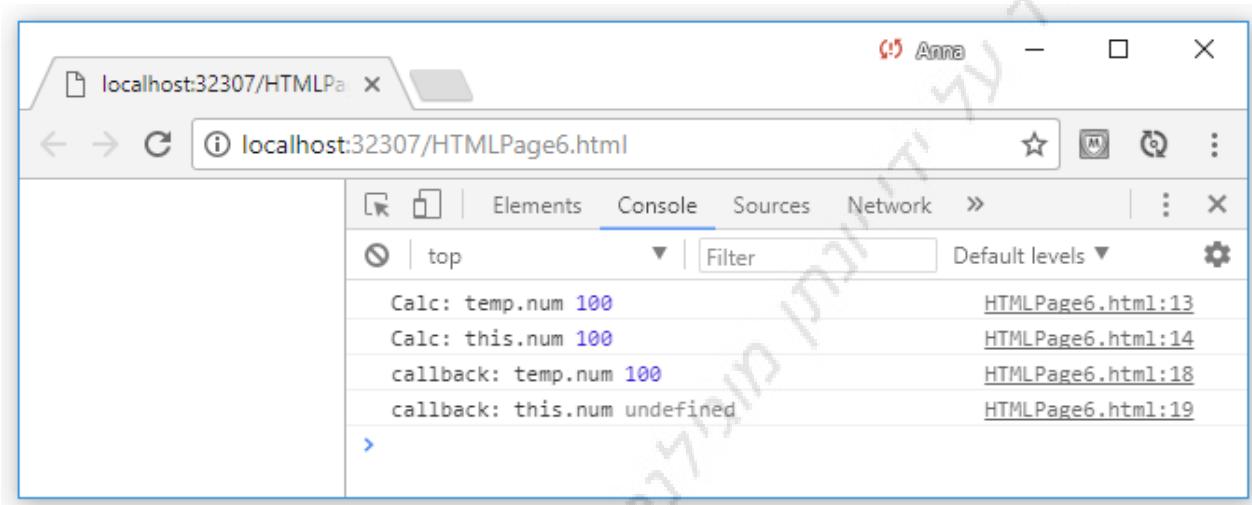


בגרסאות הקודמות, היה נהוג להתגבר על הבעיה זו בדרך הבאה:

```
<!DOCTYPE html>  
  
<html lang="en" xmlns="http://www.w3.org/1999/xhtml">  
<head>  
    <meta charset="utf-8" />  
    <title></title>  
    <script>  
        function Calc() {  
            this.num = 100;  
  
            var temp = this;  
  
            console.log("Calc: temp.num", temp.num);  
            console.log("Calc: this.num", this.num);  
  
            setTimeout(function callback() {  
  
                console.log("callback: temp.num", temp.num);  
                console.log("callback: this.num", this.num);  
  
            }, 1000);  
        }  
  
        var calc = new Calc();  
  
    </script>  
</head>
```

```
<body>
</body>
</html>
```

כasher נרץ את הקוד בדף, נקבל את התוצאה הבאה:



אולם, עם ECMAScript 2015 הדברים פשוטים יותר:

```
<!DOCTYPE html>

<html lang="en" xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta charset="utf-8" />
    <title></title>
    <script>
        function Calc() {
            this.num = 100;

            console.log(this.num);

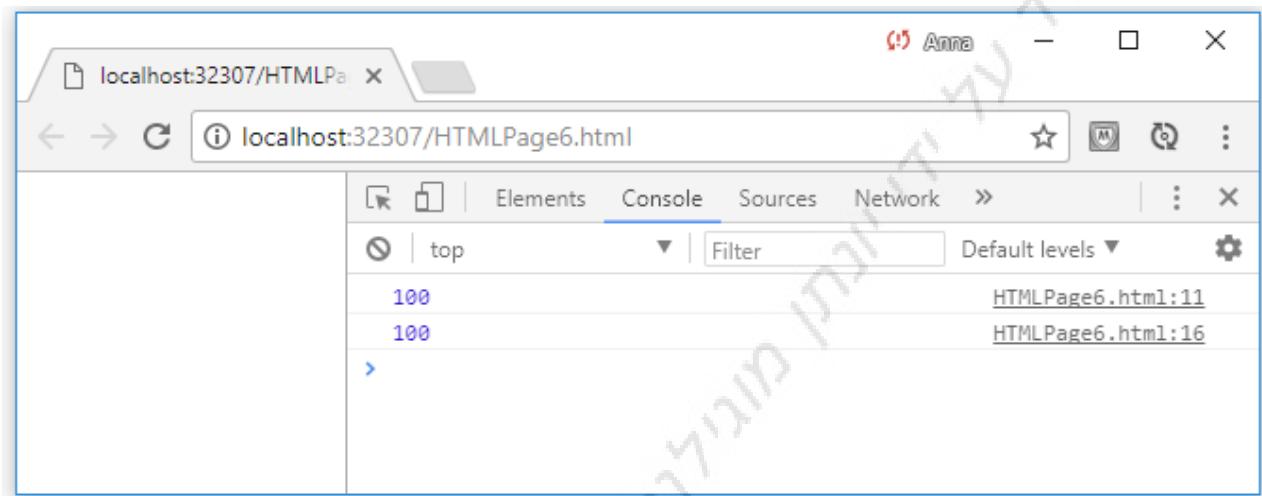
            setTimeout(() => {
                // "this" is bound to the enclosing scope's "this" value
                console.log(this.num);
            }, 1000);
        }

        var calc = new Calc();

    </script>
</head>
```

```
<body>
</body>
</html>
```

כאשר נרץ את הקוד בדפדפן, נקבל את התוצאה הבאה:



4.6. הגדרת משתנים גלובליים בפונקציה

כאשר נגדיר משתנים בפונקציה ללא המקדם const / let / var – אם הקוד לא נכתב במצב של strict mode המשטנה שנוצר מוגדר ברמה גלובלית – וונиш דרך האובייקט window גם לאחר יציאה מהפונקציה.

להלן דוגמה:

```
<!DOCTYPE html>

<html lang="en" xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta charset="utf-8" />
    <title></title>
    <script>
        var x = 123; // Context = window
        console.log(x); // 123
        console.log(window.x); // 123

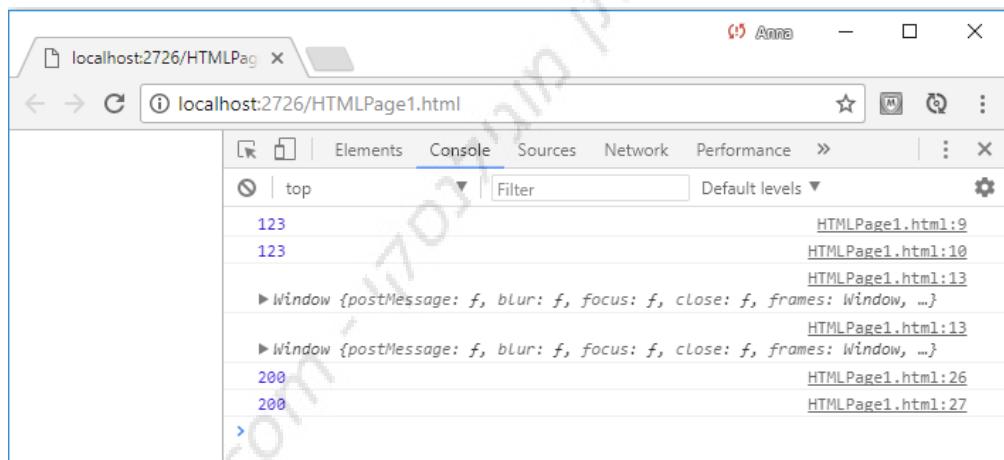
        function doSomething() { // Context = window
            console.log(this);
        }
        doSomething(); // object Window
        window.doSomething(); // object Window
```

```

function doSomethingElse() {
    var a = 100; // Private variable, not connected to the window.
    function f() { // Private function, not connected to the window.
        console.log("Hi");
    }
    b = 200; // Context = window!
}
doSomethingElse();
console.log(b); // 200
console.log(window.b); // 200
</script>
</head>
<body>

</body>
</html>

```



יש לשים לב, שהגדרת משתנים גלובליים בצורה מרומזת בתוך פונקציה, יצרו שגיאת ריצה, במצב של המוגדר בצורה הבאה:

```

(function () {
    "use strict";
    y = 200; // Not legal - will crash the script.
    alert(window.y); // Code won't get to this point.
})();

```

Self-invoking functions .4.7

הן פונקציות הקוראות לעצמן מיד בסיום הגדרתן. Immediately Invoked Function Expressions -**IIFE**

דרך ראשונה:

```
<!DOCTYPE html>

<html lang="en" xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta charset="utf-8" />
    <title></title>
    <script>

        var main = function () {
            for (var x = 0; x < 5; x++) {
                console.log(x);
            }
        }();

    </script>
</head>
<body>

</body>
</html>
```



0	HTMLPage3.html:11
1	HTMLPage3.html:11
2	HTMLPage3.html:11
3	HTMLPage3.html:11
4	HTMLPage3.html:11

דרך שנייה:

```
</html>
<!DOCTYPE html>

<html lang="en" xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta charset="utf-8" />
    <title></title>
    <script>

        (function () {
            for (var x = 0; x < 5; x++) {
                console.log(x);
            }
        })();

    </script>
</head>
<body>

</body>
</html>
```

```

</script>
</head> 0
<body> 1
</body> 2
</html> 3
        4
>

```

HTMLPage3.html:11

Closures .4.8 |

כמו ברוב שפות התכנות המודרניות - JavaScript משתמש ב-lexical scoping משמעות הדבר היא כי פונקציות מבוססות באמצעות variable scope שהיא בתוקף כאשר הם הוגדרו, ולא הם - variable scope הנמצא בתוקף כאשר הם מופעלים.

על מנת לישם את lexical scoping, המצב הפנימי של אובייקט פונקציית JavaScript חייב לכלול לא רק את קוד הפונקציה אלא גם reference ל-current scope chain. שילוב זה של אובייקט פונקציה וscope (קבוצה של bindings המשתנים של הפונקציה הם - variable scope) שביהם המצביעים על הפונקציה הנמצא בתוקף כאשר הם מופעלים, נקרא closure.

מבחינה טכנית, כל הפונקציות של JavaScript הן closures: הן אובייקטים, ויש להן chain scope הקשור בהן.

רוב הפונקציות מופעלות תוך שימוש באותו scope chain whereby הפונקציה הוגדרה. Closures הופכים מעוניינים כאשר הם מופעלים תחת scope chain שונה מזה שהיא בתוקף כאשר הם הוגדרו. מצב זה קורה בדרך כלל כאשר אובייקט פונקציה מזוהה ממהפונקציה שבתוכה הוא הוגדר. ישנן מספר טכניקות תכנותיות הכוללות את עקרון closure, והשימוש בהן געשה נפוץ יחסית בקוד JavaScript.

הצעד הראשון להבנת closures, הוא הבנת הכללים של lexical scoping עבור פונקציות מקוונות. לדוגמה, הקוד הבא:

```

<!DOCTYPE html>

<html lang="en" xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta charset="utf-8" />
    <title></title>

    <script>
        var test = "global"; // A global variable

```

```

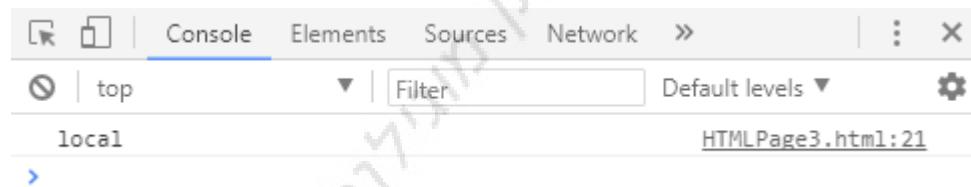
function outer() {
    var test = "local";           // A local variable
    function inner() { return test; }
    return inner()
}

console.log(outer());

</script>
</head>
<body>

</body>
</html>

```



הfonקציה החיצונית `outer()` מכירה על משתנה מקומי ולאחר מכן מגדרה ומפעילה פונקציה המחזיר את הערך של משתנה זה, וכך שפוי היא מוחזירה "local".
עשוי לשנות את הקוד, ובמקרה להפעיל את הפונקציה המקוונת על ידי הפונקציה בתוכה היא מקוונת, נחזיר את הפונקציה המקוונת בתור ערך מוחזר בכל קרייה לפונקציה `outer()`:

```

<!DOCTYPE html>

<html lang="en" xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta charset="utf-8" />
    <title></title>

    <script>
        var test = "global";      // A global variable

        function outer() {
            var test = "local";    // A local variable
            return function inner() { return test; }
        }

        var result = outer();
        console.log(result());

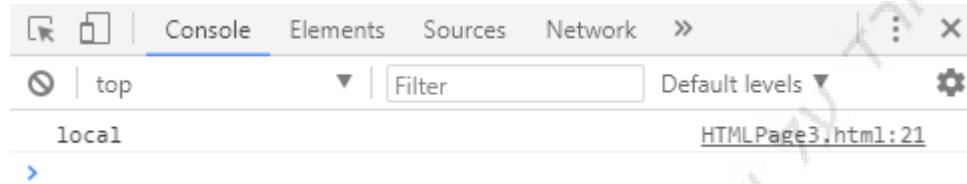
    </script>

```

```
</head>
<body>

</body>
</html>
```

כasher anou mafuilim at hafonkzia ha-pnimit machoz la-zor shvo ha-yah hogdara, nikkal at ha-totzaa ha-baah:



ha-siba la-output she-kiblmo ha-yah ha-cel basit shel lexical scoping: fonkziot JavaScript mbozutot ba-amutzot scope chain she-hita batokf casher hn hogdaro.

ha-fonkzia ha-mekonnet (inner) hogdara tacht ha-scope chain shel outer(), sheba ha-scope shel ha-meshntna hiba kshor le-urk "local". ve ha-binding hiza edin batokf casher inner() mbozut, laa meshna mahicn hoa bozut.

Zeho tebum shel closures: leshmer at ha-bindings shel ha-meshntnim ha-lokaliim v-hafematrim shel ha-fonkzia ha-chizoniit sheba hm mogdrim.

Be-slow-level programming languages como C, ha-arcitktura shel ha-shimush be-CPU ha-yah mbozutet mchsonit: am ha-meshntnim ha-lokaliim shel fonkzia mogdrim be-stack CPU, hem yofsiyu l-hatkinim casher ha-fonkzia chzra.

Abel be-JavaScript anou magdirim ha-scope chain batov reshima shel avoyikutim (la-stack of bindings). v-bekul fuim shmo-pulet fonkzia JavaScript, nozr avoyikut chadsh shnoud la-hazik at ha-meshntnim ha-lokaliim ubvor oteta kriyah, v-otetu avoyikut nosaf la-chain scope. casher ha-fonkzia chzra, ha-binding shel avoyikut hiza moser ma-ha-scope. am ain fonkziot mikonnot, azi ain uod references binding la-avoyikut v-ho yshohrr ul-id ha-garbage collector. olim, am hogdru fonkziot mikonnot, azi l-cel achat ha-fonkzia ha-alla yish references chain ha-scope, ve ha-bindings shel ha-fonkzia ha-alla ha-chizoniit. am al-la avoyikutim shel fonkzia mikonnot v-hem nasharim rak batov ha-fonkzia ha-chizoniit shlam.

Casher ha-fonkzia ha-chizoniit tshohrr ul-id ha-garbage collector hem gam yshohrru, v-la ymisico l-hacil binding la-avoyikut ha-fonkzia ha-chizoniit bha hm hogdru. Abel am fonkzia ha-chizoniit magdirah fonkzia mikonnett v-mchzirah oteta, ha-mekom shbetsu kriyah la-fonkzia ykol la-achson at ha-fonkzia ha-pnimit ha-mochzarta, azt tihia ha-tiyisot chizoniit la-fonkzia ha-mikonnet. v-lan ha-garbage collector laa yicol l-shohrr at avoyikut ha-fonkzia ha-chizoniit.

Arguments and parameters . 4.9

הגדירות פונקציית JavaScript אין מציין סוג נדרש עבור הפרמטרים של הפונקציה, ובעת קרייה לפונקציה, לא מתבצעת בדיקה כלשהי על ערכי הארגומנטים המועברים או על סוג הטיפוס שלהם. אבל כאשר פונקציה מופעלת עם ערכי ארגומנטים רבים יותר מאשר שמות פרמטרים, אין דרך להתייחס לשירותם לערכים המיותרים.

אובייקט arguments מספק פתרון לבעה זו. בתוך גוף של פונקציה, הארגומנטים שהתקבלו לפונקציה ניתנים לגישה דרך אובייקט arguments עבור אותה קרייה.

האובייקט arguments הוא אובייקט דמיי מערך המאפשר לקבל את הארגומנטים שהועברו לפונקציה כדי לאחזר לפ' מסך, ולא לפ' שם.

האובייקט arguments שימושי במספר דרכים. הדוגמה הבאה מראה כיצד ניתן להשתמש בו כדי לוודא שהפונקציה מופעלת עם המספר הרצוי של הארגומנטים, מכיוון ש- JavaScript אינו עושה זאת עבורך:

```
<!DOCTYPE html>

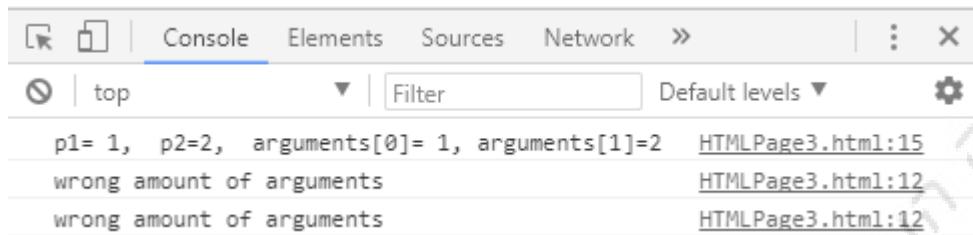
<html lang="en" xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta charset="utf-8" />
    <title></title>

    <script>
        function func(p1, p2) {

            if (arguments.length != 2) {
                console.log("wrong amount of arguments");
            }
            else {
                console.log(`p1= ${p1}, p2=${p2}, arguments[0]= ${arguments[0]}, arguments[1]=${arguments[1]}`);
            }
        }

        func(1, 2);
        func(1);
        func(1, 2, 2);
    </script>
</head>
<body>

</body>
</html>
```



The screenshot shows a browser's developer tools console tab labeled "Console". It displays three error messages: "p1= 1, p2=2, arguments[0]= 1, arguments[1]=2" at line 15 of "HTMLPage3.html", "wrong amount of arguments" at line 12 of "HTMLPage3.html", and another "wrong amount of arguments" at line 12 of "HTMLPage3.html". The "Console" tab is highlighted.

הערה: במצב Strict-mode, arguments ממענה מילה שומרה. פונקציות אינן יכולות להשתמש ב- arguments כשם פרמטר או כ משתנה מקומי, וכן לא ניתן לבצע השמה של ערכים ל arguments.



Invoking functions .4.10 |

ניתן להפעיל פונקציות JavaScript באربע דרכים:

- as functions
- as methods
- as constructors, and
- indirectly -through call() and apply()

Function Invocation

פונקציות מופעלות כפונקציות או כשיטות עם ביטוי קריאה (סוגרים). בקריאה זו, כל ביטוי וArgument (אליה בין הסוגרים) מוערך, והערכים שהתקבלו הופכים לארגומנטים של הפונקציה. ערכים אלה מוקצים לפרמטרים הנקובים בהגדרת הפונקציה. ובוגר הפונקציה, הפניה לפרמטר מעריכה את ערך הארגומנט המתאים.

עבור קריאה ל regular function, הערך המוחזר מהפונקציה הופך לערך של ה invocation expression. אם הפונקציה חוזרת מכיוון שהיא הגיעו אל סופה ללא שום פקודה המחזירה ערך, נערך המוחזר undefined. היה

Method Invocation

method הוא פונקציית המאוחסנת ב property של אובייקט. לדוגמה נוכל להגדיר method בשם `o` לאובייקט `o` באמצעות השורה הבאה:

```
o.m = function (x) { return x * x; };
```

לאחר שהגדירנו את השיטה `o` עברו האובייקט `o`. נוכל להפעיל אותה כך:

```
o.m(5);
```

שיטת הביצוע של Method שונה משיטת הביצוע של function בדרך חשובה אחת: ההקשר של ה-.`invocation context`

ביטוי גישה למאפיין אובייקט מורכבים משני חלקים:

- שם האובייקט
- שם המאפיין

כאשר המאפיין הוא method האובייקט אליו פנינו למאפיין הופך ל invocation context, וכך הפונקציה יכולה להתייחס אליו באמצעות המילה השמורה `this`. לדוגמה:

```
<!DOCTYPE html>

<html lang="en" xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta charset="utf-8" />
    <title></title>
    <script>
        var calculator = { // An object literal

            operand1: 1,
            operand2: 1,
            add: function() {

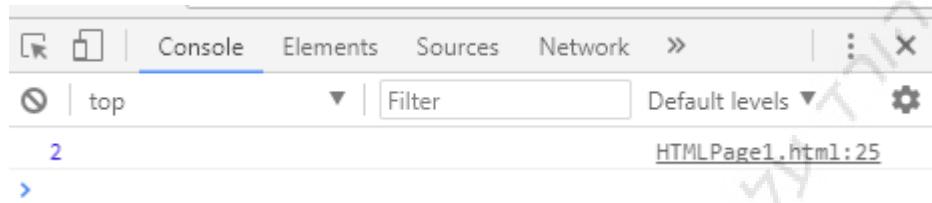
                //Note the use of the this keyword to refer to this object.
                this.result = this.operand1 + this.operand2;
            }
        };

        calculator.add(); // A method invocation

        console.log(calculator.result)

    </script>
</head>
```

```
<body>
</body>
</html>
```



המונח Methods והמילה השמורה this, מהוים אבן חשובה לפרדיגמת התכונות מונחה העצמים. כל פונקציה המשמשת כmethode מעבירה לפונקמת התכונות מונחה העצמים. כל אובייקט שדרכו היא מופעלת בדרך כלל, שיטה מבצעת איזושה פועלה על האובייקט, והתחבר של method-invocation הוא דרך אינטואיטיבית להביע את העבודה שהfonkציה שנקרה פועלת על אובייקט.
דוגמה הבאה נראה שתי שיטות לביצוע פעולה על אובייקט מסוים:

```
<!DOCTYPE html>

<html lang="en" xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta charset="utf-8" />
    <title></title>

    <script>
        var car1 = {
            color: "red",
            wheels: 4,
            setColor: function (newColor) {
                console.log("before changing car1: ", this.color);
                this.color = newColor;
                console.log("after changing car1: ", this.color);
            }
        }

        function setColor(car, newColor) {
            console.log("before changing car1: ", car.color);
            car.color = newColor;
            console.log("after changing car1: ", car.color);
        }
    </script>

```

```
//as method
car1.setColor("black");

//as function
setColor(car1, "green");

</script>
</head>
<body>
```

Message	File	Line
before changing car1: red	HTMLPage3.html	13
after changing car1: black	HTMLPage3.html	15
before changing car1: black	HTMLPage3.html	20
after changing car1: green	HTMLPage3.html	22

הfonקציות המופיעות בשתי שורות קוד אלו usableות לבצע את אותה פעולה בדיק על האובייקט car1, אך התchapir של method-invocation בשורה הראשונה מבהיר על ידי syntax שלו את הרעיון שמדובר באובייקט car1 ועליו יבוצע ה operation.

הערה: this הוא keyword, ולא משתנה או property.



תחביר JavaScript אינו מאפשר להקצות ערך ל- this. אבל שלא כמו משתנים, ל- this אין scope, ופונקציות מקוונות לא יורשות את הערכו של הפונקציה המכילה אותו.

אם nested function מופעלת כ식חה, הערכו this הוא האובייקט שהפעיל אותה. אולם אם פונקציה מקוונת מופעלת כפונקציה זו this יכול את הערך global object (במצב non-strict mode) או undefined (במצב strict mode).

זהו טעות נפוצה להניח כי פונקציה מקוונת המופעלת כפונקציה יכולה להשתמש ב- this כדי לקבל את ה- context invocation של הפונקציה החיצונית. אבל למעשה, אם רוצים לגשת לערך this של הפונקציה החיצונית, צריך לאחסן את הערך של this למשתנה אחר, כי הוא לא מוכך ב- scope של הפונקציה הפנימית.

מקובל להשתמש במשתני עזר למטרה זו. לדוגמה:

```
<!DOCTYPE html>

<html lang="en" xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta charset="utf-8" />
    <title></title>
    <script>
        var obj = {
            f1: function () {
```

```

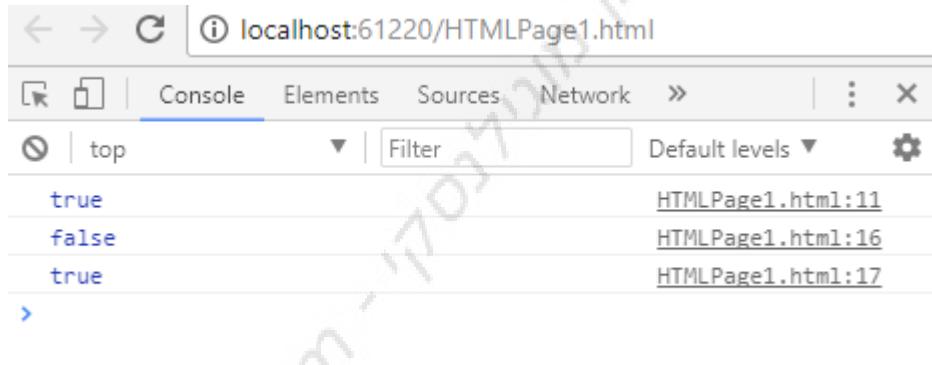
        var self = this;
        console.log(this === obj);

        f2();

        function f2() {
            console.log(this === obj);
            console.log(self === obj);
        }
    };
    obj.f1(); //Invoke the method
</script>

</head>
<body>
</body>
</html>

```



או שאנו יכולים פשוט להשתמש ב-arrow function. לדוגמה:

```

<!DOCTYPE html>

<html lang="en" xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta charset="utf-8" />
    <title></title>
    <script>
        var obj = {
            f1: function () {
                console.log(this === obj);

                var f2 = () => {
                    console.log(this === obj);
                };

                f2();
            }
        };
    </script>

```

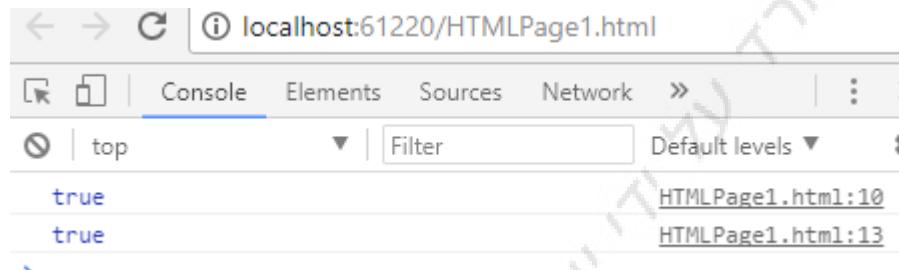
```

    obj.f1();      //Invoke the method

</script>

</head>
<body>
</body>

</html>
```



Constructor Invocation

אם לפני הפניה לפונקציה התווספה המילה `new`, אז הקראיה היא constructor invocation שמתיחס לאובייקט המכונה object prototype. לכל פונקציה ישprototype object משונה. כאשר פונקציה משתמשת כ constructor, האובייקט החדש שנוצר יורש מאפיינים מה prototype object של אותה פונקציה. invocation arguments function ב-Constructor invocations גבדלים מ method invocation ו-arguments ב- context ובערך המוחזר.

אם הפניה ל constructor - כולל רישימת ארגומנטים בסוגרים, ביטויים אלה מוערכים ומוסברים לפונקציה באותה דרך שבה הם יהיו עבור הפונקציות של פונקציות ושיטות. אבל אם ל constructor אין פרמטרים, אז התחריר מאפשר להשמיט לחלוtin את הסוגרים של הקראיה לבנאי.

לדוגמה - שתי השורות הבאות, זהות במשמעותן:

```

var o1 = new Object();
var o2 = new Object;
```

הפניה אל ה constructor יוצרת אובייקט חדש, ריק, שירש מה prototype property של הבנאי. invocation Constructor functions נועד לאתחל אובייקטים והאובייקט החדש שנוצר משמש ל context, ולכן פונקציית הבנאי יכולה להתייחס אליו עם המילה השמורה `this`.

הערה: האובייקט החדש משמש כ invocation context גם אם הפניה של הבנאי נעשית כ- method invocation
כלומר, בביטוי :

```
new o.m();
```



האובייקט ס אינו משמש כ.invocation context

Constructor functions אינן משתמשות בדרך כלל במילה return. מכיוון שתפקידם לאותחל את האובייקט החדש ולאחר מכן האובייקט הזה מוחזר בצורה implicitely return כאשר הבניי הגיע לסוף הגוף שלו. אם בנהי השתמש במפורש בהצורת return כדי להחזיר אובייקט, אז האובייקט הזה הופך להיות הערך של ביטוי 'הקריאה לבניי', אולם אם הבניי מוחזר ערך פרימיטיבי, הערך הזה לא יוחזר בפועל, אלא תבוצע החזרה של האובייקט החדש המשמש בתורו הערך של הפניה.

Indirect Invocation

פונקציות JavaScript הן אובייקטים וכמו כל האובייקטים של JavaScript יש להם שיטות.
שתי השיטות call() וapply(), מפעילות את הפונקציה בעקביפין.

שתי השיטות מאפשרות לציין במפורש את הערך this עבור הפניה, כך שאפשר להפעיל כל פונקציה כשיטה של כל אובייקט, גם אם זה לא ממש שיטה של האובייקט.

call() ו**apply()** מאפשרים להפעיל באופן עקיף פונקציה בהתאם היא שיטה של אובייקט אחר.

הארגון הראשון של apply() או call() הוא אובייקט שבו יש לבצע את הפונקציה; ארגומנט זה הוא הפונקציה f () כשיטה של האובייקט ס (לא כל ארגומנטים), ניתן להשתמש באחת משתי הפקודות הבאות:

```
f.call();
```

```
f.apply();
```

במצב ECMAScript strict mode הarginuent הראשון (call() או apply()) מייצג את הערך של this, גם אם הוא ערך פרימיטיבי או undefined. ב-ECMAScript non-strict mode (3 ו-5) ערך של null או undefined מוחלף באובייקט הגלובלי וערך פרימיטיב-אטוי מוחלף ב-object wrapper מתאים.

כל ארגומנטים שנשלחים ל- call() לאחר הארגומנט הראשון הם הערכים המועברים לפונקציה שמופעלת. לדוגמה, כדי להעביר שני מספרים לפונקציה f ולהפעיל אותה בהתאם היא שיטה של אובייקט ס, תוכל להשתמש בקוד הבא:

```
f.call(1, 2);
```

השיטה apply() דומה לשיטת call() אלא שהיא מקבלת ארגומנטים שיש להעביר לפונקציה מוגדרים כמעריך:

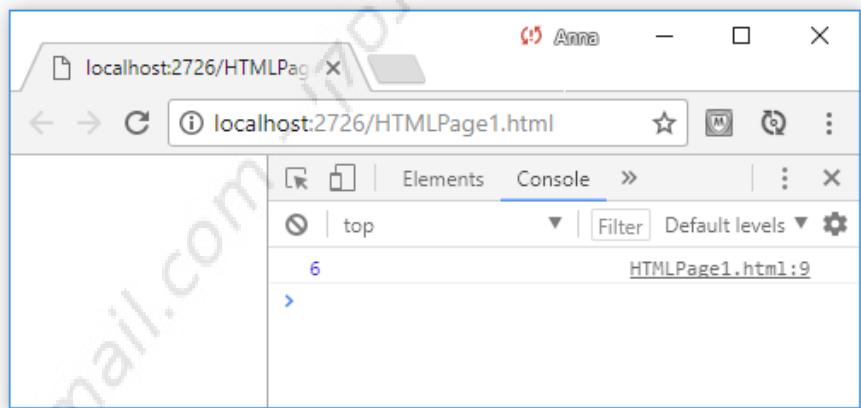
```
f.apply([1, 2]);
```

אם פונקציה מוגדרת לקבל מספר מסויים של ארגומנטים, השיטה (`apply`) מאפשרת להפעיל את הפונקציה על התוכן של מערך הארגומנטים. לדוגמה, כדי למצוא את המספר הגדול ביותר במערך של מספרים, נוכל להשתמש בשיטת (`apply`) להעביר את מרכיבי המערך לפונקציה `Math.max`:

```
<!DOCTYPE html>

<html lang="en" xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta charset="utf-8" />
    <title></title>
    <script>
        var maxNum = Math.max.apply(Math, [1, 2, 3, 4, 5, 6]);
        console.log(maxNum);
    </script>
</head>
<body>

</body>
</html>
```



לסיום, ניצור פונקציה ונקרא לה באמצעות הדריכים שסקרנו לעיל:



```
<!DOCTYPE html>

<html lang="en" xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta charset="utf-8" />
    <title></title>
    <script>
        function addSumToContext(a, b, c) {
            this.sum = a + b + c;
        }
    </script>
</head>
```

```
//as function*****
addSumToContext(10, 20, 30); // Context = window.
console.log(window.sum); // 60

//as method*****
var obj0 = {
    addSumToObj0: addSumToContext,
};
obj0.addSumToObj0(10, 20, 30); // Context = obj0
console.log(obj0.sum); // 60

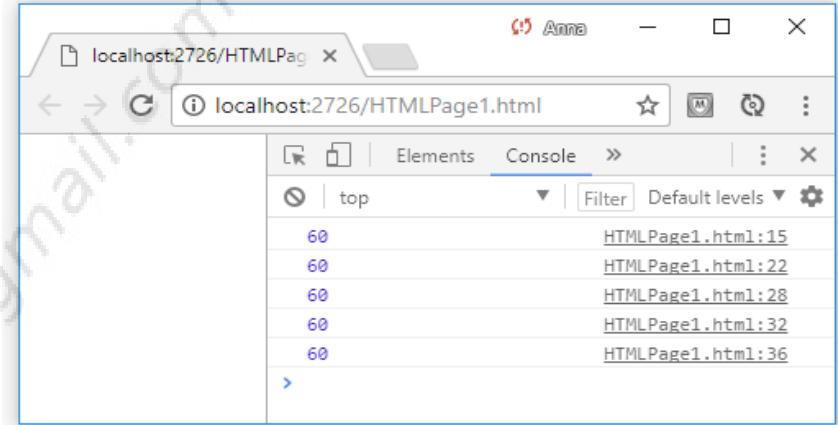
//Indirect Invocation*****
var obj1 = {};
addSumToContext.call(obj1, 10, 20, 30); // Context = obj1
console.log(obj1.sum); // 60

var obj2 = {};
addSumToContext.apply(obj2, [10, 20, 30]); // Context = obj2
console.log(obj2.sum); // 60

//as constructor*****
var obj3 = new addSumToContext(10, 20, 30); // Context = newly created obj3
console.log(obj3.sum); // 60

</script>
</head>
<body>

</body>
</html>
```



4.11. תרגילים |



תרגילים בנושא closures

תרגיל 1

נתון הקוד הבא:

```
<!DOCTYPE html>

<html lang="en" xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta charset="utf-8" />
    <title></title>

</head>
<body>
    <button id="a1">a1</button>
    <button id="a2">a2</button>
    <button id="a3">a3</button>
    <br />
    <button id="b1">b1</button>
    <button id="b2">b2</button>
    <button id="b3">b3</button>

    <script>
        for (var i = 1; i <= 3; i++) {
            var btn = document.getElementById("a" + i);
            btn.addEventListener("click", function () {
                alert (i);
            });
        }

        for (var i = 1; i <= 3; i++) {
            var btn = document.getElementById("b" + i);
            btn.addEventListener("click", function (index) {
                return function () {
                    alert(index);
                }
            }(i));
        }
    </script>
</body>
</html>
```

אם נריץ את הדף בדפסן, נקבל את הדף הבא:

a1	a2	a3
b1	b2	b3

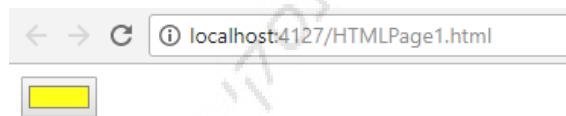
נoso לחשב (ללא הרצת הקוד – בהרצתה יבשה) מה יהיה הפלט בלחיצת על כל כפתור

תרגיל 2

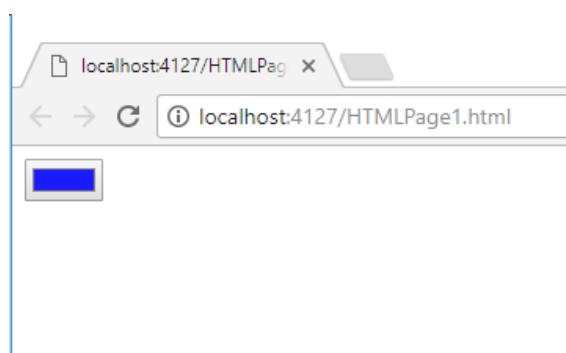
1. צרו בעמוד HTML תיבת קלט מסוג COLOR
2. צרו קוד מתאים כך שבכל בחירת צבע של הלקוח תבוצע פונקציית `setTimeOut` של `DELAY` 5000 מילישניות, הפונקציה זו תציג alert של הנתונים הבאים:
 - מספר הפעמים שהלקוח בחר צבע
 - המספר הסידורי של הבחירה זו
 - הצבע האחרון שהלקוח בחר
 - הצבע שהלקוח בחר בבחירה זו

לדוגמא:

הלקוח ביצע בחירה ראשונה של צבע צהוב



מיד לאחר מכן ביצע בחירה שנייה של צבע כחול



לאחר חמיש שניות יופיעו על המסך שתי הודעהות הבאות:



5. אובייקטים

5.1. מבנה האובייקט

אובייקט הוא **collection** properties של properties unordered collection, שלכל אחד מהם יש שם וערך. שמות properties הם מחרוזות, כך שאנו יכולים לומר כי אובייקטים מפתח מחרוזות לערכים. אובייקט JavaScript הם דינמיים, כך שניתן בדרך כלל להוסף ולמחוק מאפיינים בצורה דינמית במהלך הקוד.

בנוסף לשמירה על set של מאפיינים, אובייקט JavaScript גם יורש את המאפיינים של אובייקט אחר, המכונה ה- prototype שלו.methods של אובייקט הן בדרך כלל מוגדרים על ידי ירושה, וה- "prototypal inheritance" היא תכונה מרכזית של JavaScript.

לPROPERTY יש שם וערך. שם המאפיין יכול להיות כל מחרוזת, כולל מחרוזת ריקה, אבל לפחות לא יהיו שני מאפיינים בעלי שם זהה.

בנוסף לשם ולערך, לכל PROPERTY יש ערכים משוכחים שנקראים:

- התכונה writable מצינית אם ניתן לעורר את ערך המאפיין.
- התכונה enumerable מצינית אם שם המאפיין מוחזר על ידי לולאת for/in loop.
- התכונה configurable קובעת אם ניתן למחוק את המאפיין ואם תכונתו ניתנת לשינוי.

בנוסף למאפיינים שלה, לכל אובייקט יש שלוש object attributes הקשורים אליו:

- אב טיפוס של אובייקט - object prototype הוא הפניה לאובייקט אחר שמננו האובייקט הנוכחי יורש את המאפיינים.
- מחרוזת המסובגת את סוג האובייקט.
- דגל של האובייקט המציין אם ניתן להוסיף מאפיינים חדשים לאובייקט. extensible flag

Prototypes

לכל אובייקט JavaScript יש אובייקט JavaScript שני (או null, אבל זה נדיר) המשויך אליו. האובייקט השני ידוע כаб טיפוס, והאובייקט הראשון יורש מאפיינים מבית הטיפוס.

כל האובייקטים שנוצרו על ידי object literals יש את אותו אובייקט אב טיפוס, אנחנו יכולים להתייחס לאובייקט אב הטיפוס הזה בקוד כמו Object.prototype.

Object.prototype הוא אחד האובייקטים הנדרים שאין להם אב טיפוס: הוא אינו יורש מאפיינים כלשהם. לכל שאר ה built-in constructors יש אובייקט אב טיפוס.
לדוגמה, Date.prototype יורש מאפיינים מ- Object, קר שאובייקט Date שנוצר על-ידי new() יורש מאפיינים משני אובייקטים - מ Date.prototype ו- Object.prototype. סדרה מקושרת זו של אובייקטים אבטיפוסים ידועה כ chain prototype.

שלוש קטגוריות של אובייקטי JavaScript:

- **native object** - הוא אובייקט או סוג של אובייקטים המוגדרים על פי מפרט ECMAScript. לדוגמה: מערכים, פונקציות, ותאריכים.
- **-host object** - הוא אובייקט שהוגדר על ידי host environment (כגון דפדפן אינטרנט) שבו מוטבע JavaScript. אובייקט HTMLElement המיצגים את המבנה של דף אינטרנט ב- host objects. host objects הם host objects העשויים גם להיות native objects, לדוגמה כאשר host environment מגדרה שיטות שהן אובייקטי פונקציית JavaScript רגילים.
- **user-defined object** - אובייקט המוגדר על ידי המשתמש הוא אובייקט שנוצר על ידי ביצוע קוד JavaScript.

שני סוגי של properties

- **own property** - הוא מאפיין המוגדר ישירות באובייקט.
- **inherited property** - הוא מאפיין שהוגדר על ידי אובייקט אב הטיפוס של אובייקט.

5.2. יצירת אובייקט

ישנן שלוש דרכים ליצור user-defined object:

- ע"י object literal
- ע"י new
- ע"י ()Object.create

בפרק זה נרחיב על כל אחד מהאופנים הנ"ל.

Object Literals

הדרך הקלה ביותר ליצור אובייקט היא על ידי object literal. object literal הוא רשיימה בתוך סוגרים מסולסלים של מפתחות וערכים המופרדים בפסיקים ביניהם. הנה כמה דוגמאות:

המונח object הוא ביטוי שיציר ומתחחל אובייקט חדש ונבדל בכל פעם שהוא מתרבע. הערך של כלocco מופיע בכל פעם המילולית מוערכת. משמעו הדבר היא כי אובייקט יחיד של object literal יכול ליצור אובייקטים חדשים רבים אם הוא מופיע בתוך הגוף של לולאה בפונקציה הנקראת שוב ושוב, וכי ערכי המאפיינים של אובייקטים אלה עשויים להיות שונים זה מזה.

JavaScript Object-literal Improvements

ES2015 הוסיף את השיפורים הבאים:

```
<!DOCTYPE html>

<html lang="en" xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta charset="utf-8" />
    <title></title>
    <script>
        function getDynamicKey() {
            return 'some key';
        }

        let sameVal = "test";

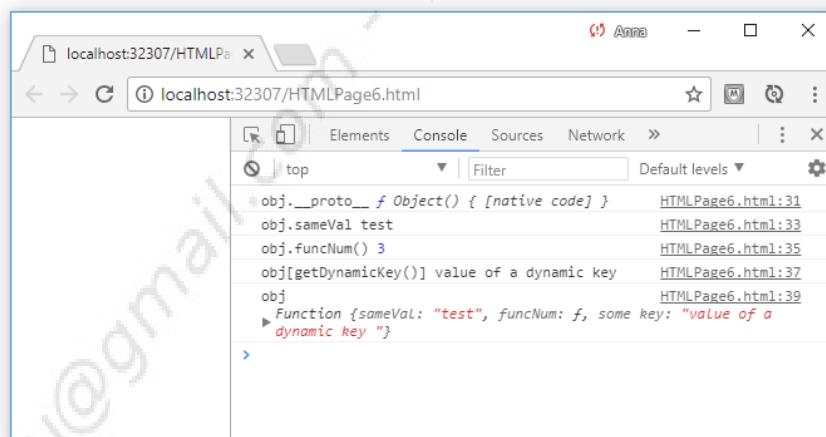
        let obj = {
            // Prototypes can be set this way
            __proto__: Object,
            // key === value, shorthand for sameVal: sameVal
            sameVal,
        }
    </script>
</head>
<body>
    <h1>Hello world!</h1>
</body>
</html>
```

```
// Methods can now be defined this way
funcNum() {
    return 3;
},
// Dynamic values for keys
[getDynamicKey()]: 'value of a dynamic key '
};

console.log("obj.__proto__",obj.__proto__);
console.log("obj.sameVal",obj.sameVal);
console.log("obj.funcNum()",obj.funcNum());
console.log("obj[getDynamicKey()]", obj[getDynamicKey()]);
console.log("obj", obj);

</script>
</head>
<body>

</body>
</html>
```



לעומת זאת, הדרך החדשה מושגת את הדרך הארכאה הבאה:

```
<!DOCTYPE html>

<html lang="en" xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta charset="utf-8" />
```

```

<title></title>
<script>
    function getDynamicKey() {
        return 'some key';
    }

    var sameVal = "test";

    var obj = {
        sameVal: sameVal,
        funcNum: function () {
            return 3;
        }
    };

    obj.prototype = Object;
    obj[getDynamicKey()] = 'value of a dynamic key';

    console.log("obj.prototype", obj.prototype);

    console.log("obj.sameVal", obj.sameVal);

    console.log("obj.funcNum()", obj.funcNum());

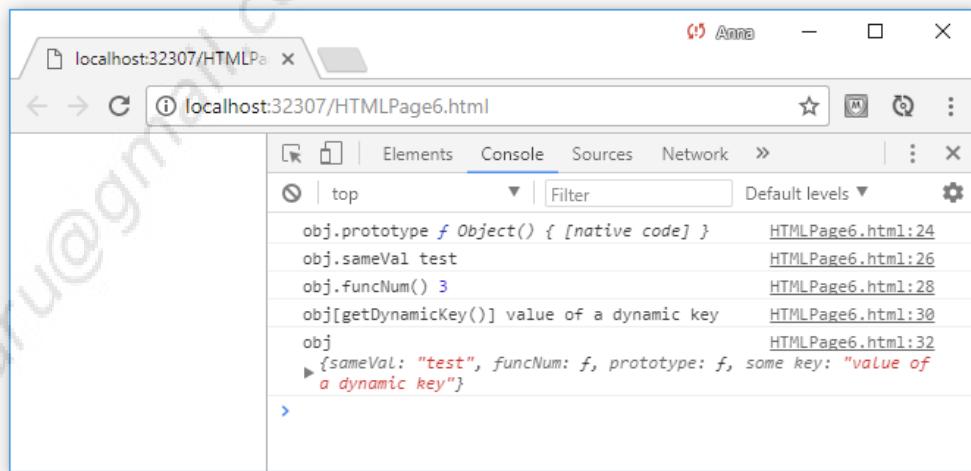
    console.log("obj[getDynamicKey()]", obj[getDynamicKey()]);

    console.log("obj", obj);

</script>
</head>
<body>

</body>
</html>

```



יצירת אובייקט על ידי new

האופרטור `new` יוצר ומתחול אובייקט חדש.

על המילה `new` להיות מלאה בפניה לפונקציה. פונקציה המשמשת בדרך זו נקראת בנה ומשמשת לתחול אובייקט חדש.

כוללת built-in constructors types native עבור Core JavaScript. לדוגמה:

```
<!DOCTYPE html>

<html lang="en" xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta charset="utf-8" />
    <title></title>
    <script>
        var o = new Object(); // Create an empty object: same as {}.
        var a = new Array(); // Create an empty array: same as [].
        var d = new Date(); // Create a Date object representing the current time
    </script>
</head>
<body>
</body>
</html>
```

יצירת אובייקט על ידי Object.create()

מגדיר שיטה 5 ECMAScript Object.create(), שיצרת אובייקט חדש, באמצעות הארגומנט הראשון שלה CAB טיפוס של אובייקט זה.

גם לוקחת ארגומנט שני אופציוני המתאר את המאפיינים של האובייקט החדש.

Object.create היא פונקיה סטטית, ולא שיטה המופעלת על אובייקטים בודדים. וכדי להשתמש בה, יש להעביר את האובייקט CAB טיפוס הרצוי:

```
var o1 = Object.create({ x: 1, y: 2 }); // o1 inherits properties x and y.
```

אפשר לשלוח את זהה כארגומנט, כדי ליצור אובייקט חדש שאין לו CAB טיפוס, אבל האובייקט החדש שנוצר לא יירש כלום, אפילו לא שיטות בסיסיות כמו `toString`:

```
var o2 = Object.create(null); // o2 inherits no props or methods.
```

אם נרצה ליצור אובייקט ריק רגיל (כמו האובייקט המוחזר על ידי {} או new Object()), נعتبر כארוגומנט את הערך Object.prototype:

```
var o3 = Object.create(Object.prototype); // o3 is like {} or new Object().
```

3.5. קראת המאפיינים ושינוי האובייקט

כדי לקבל את הערך של מאפיין, יש להשתמש בנקודה (.) או בסוגרים מרובעים ([]).

אם נשתמש בנקודה, המילה הימנית חייבת להיות מזזה פשוט של אחד ממאפייני האובייקט. אם משתמשים בסוגרים מרובעים, הערך בתוך סוגרים חייב להיות ביטוי המחזיר מחרוזת המכילה את שם המאפיין הרצוי:

```
<!DOCTYPE html>

<html lang="en" xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta charset="utf-8" />
    <title></title>
    <script>
        var car = {
            color: "white",
            name:"Tommi"
        }

        var color = cat.color;           // Get the "color" property of the cat.
        var name = cat["name"]          //Get the "name" property of the cat.

    </script>
</head>
<body>

</body>
</html>
```

Inheritance

אובייקט JavaScript מכילים קבוצה של "own properties", והם גם יורשים קבוצה של מאפיינים מאובייקט אב הטיפוס שלהם.

נניח שננסה לקרוא את התוכן של מאפיין x באובייקט o, אם לו אין מאפיין משלו עם שם זה, אז יתבצע חיפוש באובייקט אב טיפוס של o עבור x.

אם לאובייקט אב טיפוס אין מאפיין משלו בשם זה, אבל יש לו אב טיפוס עצמו, החיפוש יתבצע על אב הטיפוס של אב הטיפוס. פעולה זו נמשכת עד שימצא ה x או עד שייררך חיפוש של אובייקט עם אב טיפוס ריק.

של אובייקט יוצר שרשרת או רשימה מקוشرת שממנה prototype attribute יונקת לראות, הירשימים מאפיינים.

```
var o = {} // o inherits object methods from Object.prototype
o.x = 1;    // and has an own property x.

var p = inherit(o);      // p inherits properties from o and Object.prototype
p.y = 2;    // and has an own property y.

var q = inherit(p);      // q inherits properties from p, o, and
Object.prototype
q.z = 3;    // and has an own property z.

var s = q.toString(); // toString is inherited from Object.prototype
q.x + q.y // => 3: x and y are inherited from o and p
```

עכשו נניח שנרצה להשים ערך לתוך המאפיין x של אובייקט o. אם לו כבר יש מאפיין משלו (noninherited) בשם x, אז ההשמה פשוט משנה את הערך של מאפיין קיים זה. אחרת, ההשמה יוצרת מאפיין חדש בשם x באובייקט o. אם o קיבל בירושה בעבר את המאפיין x, המאפיין בירושה מושתר כתעל ידי המאפיין עצמו שנוצר באותו שם.

```
var o = { r: 1 }; // An object to inherit from
var c = inherit(o); // c inherits the property r
c.x = 1; c.y = 1; // c defines two properties of its own
c.r = 2;          // c overrides its inherited property
o.r;             // => 1: the prototype object is not affected
```

יש מקרה חריג אחד לכל הגדרת מאפיין באובייקט המקורי. והוא אם ס. יורש את המאפיין א', ואוטו מאפיין הוא מאפיין setter method עם accessor, אזsetter method נקרא במקום ליצור מאפיין חדש א' ב-0.

Deleting Properties

האופרטור delete מסיר מאפיין מאובייקט.

הօפרנד של האופרטור delete צריך להיות ביטוי גישה למאפיין. המחקה פועלת על המאפיין עצמו:

```
var car = {
    color: "white",
    name: "Tommi"
}
delete cat.color;
delete cat["name"];
```

האופרטור delete מוחק רק את המאפיינים של האובייקט עצמו, ולא את תכונות שנוספו בירושה. (כדי למחוק מאפיין בירושה, יש למחוק אותו מאובייקט אב-טיפוס שבו הוא מוגדר. הדבר משפיע על כל אובייקט שירש מאב-טיפוס זה).

מחיקת הביטוי ממחירה את הערך- true אם המחקה הצלחה או false אם למחיקה לא הייתה השפעה (כגון מחיקת מאפיין שאין קיים) delete אינו מסיר מאפיינים בעלי configurable attribute בעל הערך false.

אופרטור in

כדי לבדוק אם לאובייקט יש מאפיין עם שם נתון. ניתן להשתמש באופרטור חוץ, או בשיטות:

- hasOwnProperty ()
- propertyIsEnumerable ()

אופרטור חוץ מקבל משMAIL את שם מאפיין (בטור מחרוזת) ומימין – את האובייקט שරוצים לבדוק האם המאפיין קיים בו ואז יחזיר true אם לאובייקט יש מאפיין פרטי או מאפיין בירושה בשם זה:

```
<!DOCTYPE html>

<html lang="en" xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta charset="utf-8" />
    <title></title>
    <script>
        var o = { x: 1 }
```

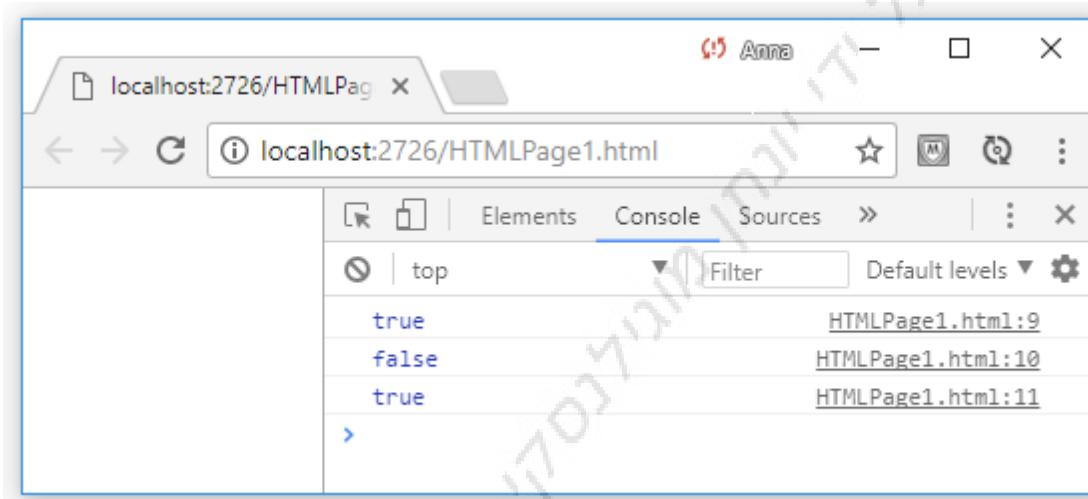
```

        console.log("x" in o); // true: o has an own property "x"
        console.log("y" in o); // false: o doesn't have a property "y"
        console.log("toString" in o); // true: o inherits a toString
        property

    </script>
</head>
<body>

</body>
</html>

```



השיטה `hasOwnProperty()` בודקת אם לאובייקט יש מאפיין מסוים עם השם הנתון. היא מחזירה `false` עבור תכונות בירושה:

```

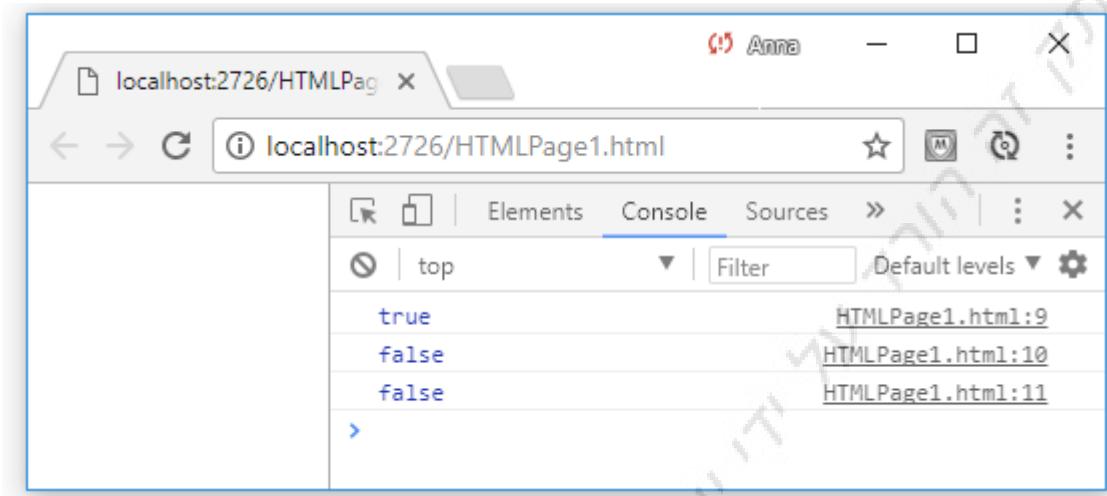
<!DOCTYPE html>

<html lang="en" xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta charset="utf-8" />
    <title></title>
    <script>
        var o = { x: 1 }
        console.log(o.hasOwnProperty("x")); // true: o has an own property x
        console.log(o.hasOwnProperty("y")); // false: o doesn't have a property y
        console.log(o.hasOwnProperty("toString")); // false: toString is an
        inherited property

    </script>
</head>
<body>

</body>
</html>

```



Enumerating Properties

בכדי להציג רשימה של כל המאפיינים של אובייקט. נעשה בדרך כלל שימוש בלולאת `for/in`.

לולאה `for/in` מפעילה את גוף הלולאה פעם אחת עבור כל מאפיין (`own` או `inherited`) של האובייקט שצויין, ומקצה את שם המאפיין למשתנה לולאה. שיטות מובנות שאובייקטים ירש לא יופיעו בczורה זו. לדוגמה:

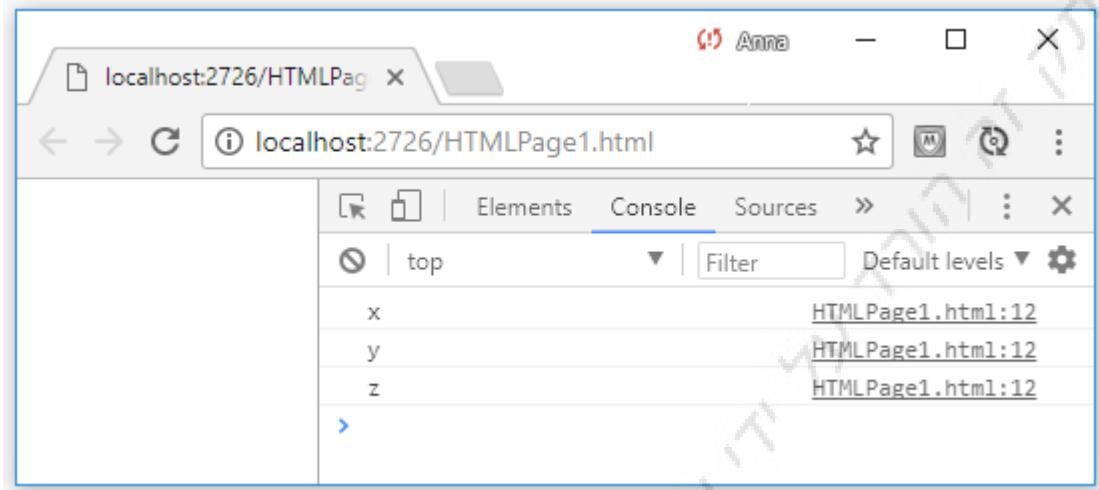
```
<!DOCTYPE html>

<html lang="en" xmlns="http://www.w3.org/1999/xhtml">
<head>
  <meta charset="utf-8" />
  <title></title>
  <script>

    var o = { x: 1, y: 2, z: 3 }; // Three enumerable own properties
    o.propertyIsEnumerable("toString") // => false: not enumerable
    for (p in o) // Loop through the properties
      console.log(p); // Prints x, y, and z, but not toString

  </script>
</head>
<body>

</body>
</html>
```



בנוספּ לולאת `for/in`, ECMAScript 5 מגדרה שתי פונקציות המחזירות את שמות מאפיינים. הראשונה היא `(Object.keys()`, המחזירה מערך של המאפיינים מסווג `own properties` של האובייקט. השנייה היא `(Object.getOwnPropertyNames()`. שפעלה כמו `(Object.keys()` אבל מחזירה את השמות של כל ה `own properties` של האובייקט שצוין, לא רק.

Property Getters and Setters

אמרנו כי מאפיין אובייקט הוא שם, ערך וקבוצת תכונות. ב- ECMAScript 5 ניתן להחליף את הערך בשיטה אחת או שתים, הידועים בשם `getter` ו- `setter`.

מאפיינים שהוגדרו על ידי `getters` ו- `setters` ידועים לעיתים כ `accessor properties`.

כאשר תוכנית ניגשת לקרוא את הערך של `accessor property` מפעילה את שיטת `getter` (לא מקבלת שום ארגומנטים). הערך המוחזר של שיטה זו הופך לערך של ביטוי הגישה למאפיין. כאשר תוכנית עורכת את הערך של `accessor property` היא מפעילה את שיטת `setter`, ומעבירה את הערך של הצד הימני של ההשמה.

ל מאפיין של קראיה / כתיבה, `Accessors properties` אין `writable attribute` כמאפייני נתונים. אם מאפיין מכיל `getter` ו- `setter`, זה המאפיין של קראיה / כתיבה.

הדרך הקלה ביותר להגדיר `Accessors properties` היא עם `object literal syntax`.

```
<!DOCTYPE html>
<html lang="en" xmlns="http://www.w3.org/1999/xhtml">
<head>
  <meta charset="utf-8" />
  <title></title>
```

```

<script>

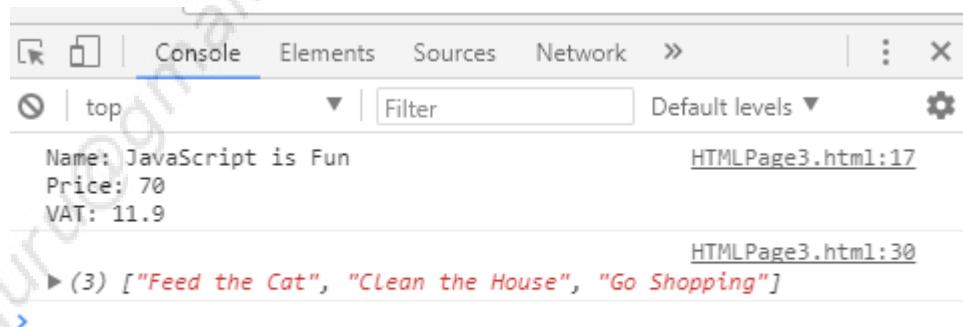
    // ----- Object get Function: -----
    var book = {
        name: "JavaScript is Fun",
        price: 70,
        get vat() {
            return this.price * 0.17;
        },
    };
    console.log("Name: " + book.name + "\nPrice: " + book.price + "\nVAT: " +
book.vat);

    // ----- Object set Function: -----
    var tasks = {
        all: [],
        set todo(task) {
            this.all.push(task);
        }
    }
    tasks.todo = "Feed the Cat";
    tasks.todo = "Clean the House";
    tasks.todo = "Go Shopping";
    console.log(tasks.all);

</script>
</head>
<body>

</body>
</html>

```



Property Attributes

בנוספ' לשם ולערך, למאפיינים יש attributes המציגנים הרשות עברו: written, enumerated, configured ב-ECMAScript 5 אין אפשרות להגדיר תכונות אלה: כל המאפיינים שנוצרו על-ידי תוכניות 3 ECMAScript ניתנים כתיבה, להגדלה, ולקבליה על ידי חוץ, ואין אפשרות לשנות זאת. סעיף זה מסביר את ה-API של ECMAScript 5 להגדרת attributes של המאפיינים.

שיטות ה-ECMAScript 5 לקריאת וקבעת המאפיינים של מאפיין משתמש באובייקט הנקרא property descriptor כדי לייצג את קבוצת ארבע התכונות. אובייקט property descriptor כולל מאפיינים בעלי אותו שם כמו התכונות של המאפיין, והם:

value, writable, enumerable, and configurable

ל-attributes של property descriptor יש get and set properties וכך אפשר לקבל ולשנות מאפיינים כדי לקבל את ה-property descriptor בעל שם של אובייקט מסוים, אפשר לרשום:

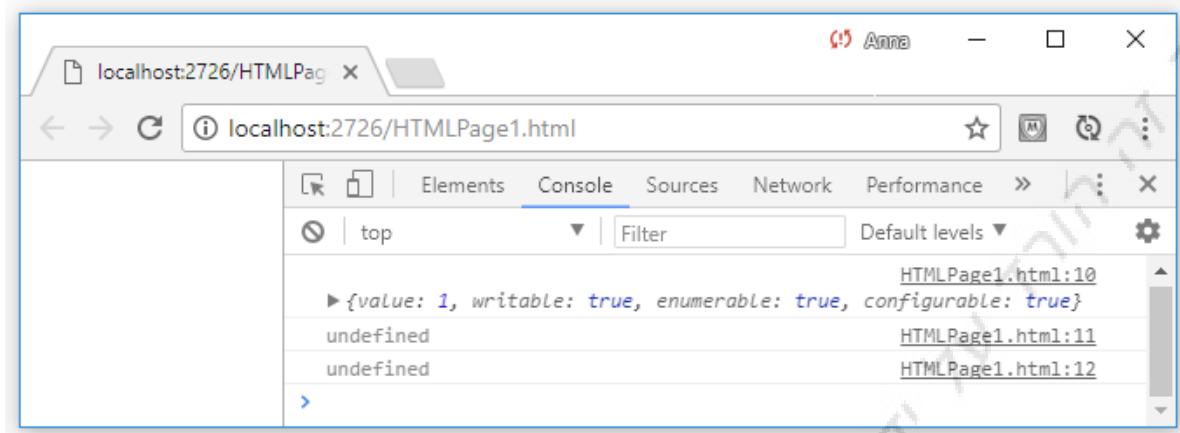
```
<!DOCTYPE html>

<html lang="en" xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta charset="utf-8" />
    <title></title>
    <script>

        var o = { x: 1, y: 2, z: 3 }; // Three enumerable own properties
        console.log(Object.getOwnPropertyDescriptor(o, "x"));
        console.log(Object.getOwnPropertyDescriptor(o, "t")); // undefined, no
such prop
        console.log(Object.getOwnPropertyDescriptor({}, "toString")); // undefined,
inherited

    </script>
</head>
<body>

</body>
</html>
```



כדי להגדיר את ה attributes של מאפיין, או כדי ליצור מאפיין חדש עם attributes מוגדרים, יש להשתמש בפונקציה `Object.defineProperty()`, המקבילה את הפרמטרים הבאים:

- פרמטר ראשון: שם האובייקט
- פרמטר שני – שם המאפיין שרצים ליצור
- פרמטר שלישי – עבור המאפיין שיצרים

:לדוגמה:

```

<!DOCTYPE html>

<html lang="en" xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta charset="utf-8" />
    <title></title>
    <script>

        var o = {}; // Start with no properties at all

        //Add a nonenumerable data property x with value 1.
        Object.defineProperty(o, "x", {
            value: 1, writable: true, enumerable: false, configurable: true
        });

        // Check that the property is there but is nonenumerable
        console.log(o.x); //      =>      1
        console.log(Object.keys(o)); //=>      []

        //Now modify the property x so that it is read-only
        Object.defineProperty(o, "x", { writable: false });

        //Try to change the value of the property
    </script>

```

```

o.x = 2;      // Fails silently or throws TypeError in strict mode

console.log(o.x); // => 1

//The property is still configurable, so we can change its value like this:
Object.defineProperty(o, "x", { value: 2 });

console.log(o.x); // => 2

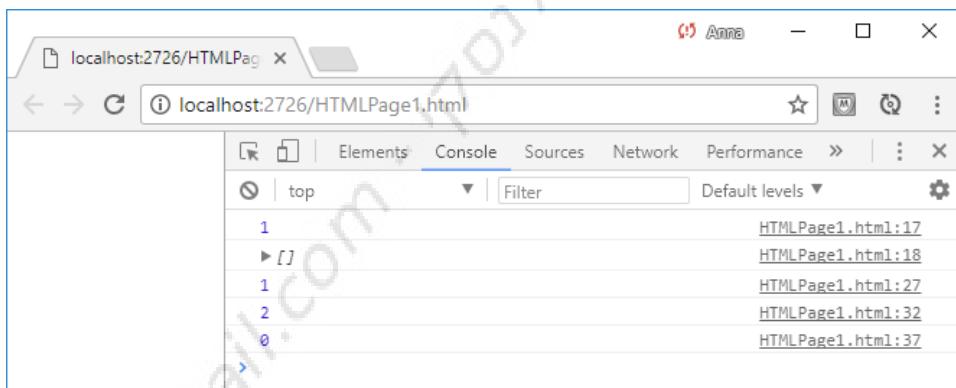
//Now change x from a data property to an accessor property
Object.defineProperty(o, "x", { get: function () { return 0; } });

console.log(o.x); // => 0

</script>
</head>
<body>

</body>
</html>

```



תרגיל בנושא אובייקטים



1. צרו אובייקט המתאר סטודנט ומכל:
 - שם פרטי
 - שם משפחה
 - כתובות
 - מערכ ציונים ריק
 - פונקציית `setter` המוסיפה ציון חדש למערך הציונים.
 - פונקציית `getter` המחזיר את ממוצע הציונים.

- פונקציית `toString` המחזירה ע"י `return` (ולא מציגה ע"י `alert`) את כל הפרטים של הסטודנט כמחרוזת אחת.
- .2. הוסיף מספר ציונים ע"י פונקציית ה-`setter`.
- .3. קיראו לפונקציית `to` של האובייקט והציגו את המחרוזת המוחזרת ע"י `alert`.

6. אסינכרוני

תכנות סינכרוני אומר שם שורה קוד 1 כתובה לפני שורה קוד 2, עד ששורות קוד 1 לא תסתיים לרווח, שורה קוד 2 לא תפעל.

אפשר לתאר את זה כמו תור לקניית כרטיסים. עד שהאדם לפניו בתור לא יסיט, לא יוכל לקנות כרטיס עצמו, והוא יתurning לאדם אחר.

לדוגמה :

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>Document</title>
</head>
<body>

<script>
    console.log("I run first!");
    console.log("I run second!");

</script>

</body>
</html>
```

בתכנות אסינכרוני, יכולות להיות 2 שורות של קוד, שורה קוד 1 מתואמת לרווח בהתאם לפעולה שתקרה בעתיד, ואז שורת קוד 2 תפעל לפניה.

אפשר לתאר את זה כמו הזמן במסעדה : שולחן 1 מזמן מנה, ואחריו שולחן 2 מזמן מנה. במידה והמנה של שולחן 2 מוכנה (המנה של שולחן 1 לקחה יותר זמן לבישול) שולחן 2 יקבל את המנה שלו, בלי צורך להמתין למנה של שולחן 1. כמובן הגעת המנה מתואמת לאירוע שקרה (המנה מוכנה) ללא תלות בסדר הפעולות.

לאורך השנים פותחו בג'אווה סקריפט מספר דרכים שבהם נוכל לזמן חלקי קוד, להפעלה בעקבות פעולה שתואמה לעתיד.

הדרך הראשונה היא `callback function`

Callback function . 6.1 |

ב זו פונקציות הן `first class objects`, כלומר פונקציה היא מטיפוס של `object`. זה נותן לנו את יכולת לאחסן פונקציות בתוך משתנים, להחזיר פונקציה בתור `return value` מפונקציה אחרת, וכן להעביר פונקציה בתור ארגומנט לפונקציה אחרת.

פונקציית `callback` היא פונקציה שעוברת בתור ארגומנט לפונקציה אחרת, ואז `execute` בתור אותה הפונקציה.

לדוגמה :

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>Document</title>
</head>

<body>
    <button id="btn">click</button>
    <script>
        document.getElementById("btn").addEventListener("click", function myNameIsCallback() {
            alert("I am the code that execute inside the callback function!");
        });
        console.log("I run first!");

    </script>
</body>

</html>
```

שימוש לב לקוד : הפונקציה `addEventListener` היא עצמה פונקציה (שימוש לב לסוגרים () כשאנחנו קוראים לה) לתוכה אנחנו מעבירים 2 ארגומנטים, הראשון הוא מחזרות "click" (שאומרת להאזין לאירוע של click), הארגומנט השני הוא פונקציה. שימוש לב שהקוד של הפונקציה `execute` אחרי הקוד שיש בתוך `console.log` , למרות שלפי סדר השורות הוא מופיע קודם . זה בכלל שהפונקציה מתזמנת לרוץ רק אחרי שקורה אירוע של `click`. הפונקציה הזאת היא פונקציית `callback`, שעוברת בתוך ארגומנט לפונקציה אחרת, ו`execute` בצורה אסינכרונית, ללא תלות ברכף השורות.

במשך שנים היה מקובל להשתמש בג'אווה סקריפט אך ורק ב `callback function` בשביל לבצע פעולות אסינכרניות. הבעייה הייתה שבגלל אופי השפה והשימוש הנרחב בפעולות אסינכרניות, קוד שעשה שימוש נרחב `callback` , הופך להיות לא קרייא ולא קל לתחזקה. כפי שיכירנו לראות לדוגמה בקטע קוד הבא :

```
fs.readdir(source, function (err, files) {
  if (err) {
    console.log('Error finding files: ' + err)
  } else {
    files.forEach(function (filename, fileIndex) {
      console.log(filename)
      gm(source + filename).size(function (err, values) {
        if (err) {
          console.log('Error identifying file size: ' + err)
        } else {
          console.log(filename + ' : ' + values)
          aspect = (values.width / values.height)
          widths.forEach(function (width, widthIndex) {
            height = Math.round(width / aspect)
            console.log('resizing ' + filename + 'to ' + height + 'x' + height)
            this.resize(width, height).write(dest + 'w' + width + '_' + filename, function(
              if (err) console.log('Error writing file: ' + err)
            })
          }).bind(this))
        }
      })
    })
  }
})
```

וגරר אחריו מושג שנקרא call back hell

לכן נוצר הצורך ליצור כלי יותר נוח בשפה שביל שימוש בפעולות אסינכרוניות. הכלי הזה נקרא Promise.

Promises .6.2 |

פרומיס הוא אובייקט שאמור להכיל ערך בזמן קלשו בעתיד. כלומר עתיד להיות fillfull בצורה אסינכרונית. זהו אובייקט המתאר השלמת הפעולה האסינכרונית, או במקורה של שגיאה - דחיתת הפעולה.

פרומיס יש שלושה מצבים :

הראשון הוא Pending - שאומר שהערך של הפרומיס עדין לא הוחלט. כלומר הפרומיס עדין מכהה לסיום של פעולה אסינכרונית.

השני הוא Fullfield - קורה מיד בסיום הפעולה האסינכרונית, והשלמת הזנת הערך בצלחה לתוך הפרומיס.

השלישי הוא Rejected - הפעולה האסינכרונית נכשלה, והפרומיס לעולם לא יהיה fillfil

כאשר פרומיס הוא במצב של Pending , הוא אמור לשנות במצב של Fullfield או של Rejected, אבל אחרי שהוא שונה לאחד המצביעים האלה, הוא לא יוכל יותר לשנות את המצב שלו לעולם.

דוגמת קוד לשימוש ב :promise

```
let promise1 = new Promise(function (resolve, reject) {
  setTimeout(() => resolve("done!"), 1000);
});

let promise2 = new Promise(function (resolve, reject) {
  reject(new Error("Error"));
});

promise1.then((result => {
  console.log(result);
})).catch(err => {
  console.error(err);
})

promise2.then((result => {
  console.log(result);
})).catch(err => {
  console.error(err);
})
```

פרומייס אומנם היוו דרך יותר נוחה לשימוש בקוד אסינכרוני, אבל לאורך השניים גם שימוש נרחב בפרומייס הפך את הקוד לקשה לקריאה ותחזוקה, כמו שניתן לראות בקטע קוד הבא:

```
getTweetsFor("domenic") // promise-returning async function
  .then(function (tweets) {
    var shortUrls = parseTweetsForUrls(tweets);
    var mostRecentShortUrl = shortUrls[0];
    return expandUrlUsingTwitterApi(mostRecentShortUrl); // promise-returning async function
  })
  .then(doHttpRequest) // promise-returning async function
  .then(
    function (responseBody) {
      console.log("Most recent link text:", responseBody);
    },
    function (error) {
      console.error("Error with the twitterverse:", error);
    }
  )
```

ולכן הוחלט לפתח בג'אווה סקרייפט מנגנון אפילו יותר נוח לשימוש בקוד אסינכרוני בשם :async await :

Async await .6.3 |

הruk ליצירת המנגנון של `await` בשפה הוא שימוש בקוד קרייא יותר. הקוד הוא אסינכרוני לכל דבר, מלבד העובדה שהוא כתוב כמו קוד סינכרוני, שורה אחר שורה. מה שמייצר דרך לכתיבה קוד אסינכרוני קרייא ונוח יותר לתחזוקה.

ונכל להבין אותו בצורה הכי טובה באמצעות שימוש בדוגמאות:

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>Document</title>
</head>

<body>
    <script>
        async function asyncFunction() {
            return "value";
        }

        asyncFunction().then(alert)
    </script>
</body>
</html>
```

את המילה השמורה `chuya` נשים בתחילת הפונקציה. מה שהופך את הפונקציה ל - `async .function`

כפי שאפשר לראות בדוגמה, בשביל להמתין לערך החזר מהפונקציה משתמש ב `then`, בדיק כמה השימוש שעשינו עם `Promise`.

ונכל גם להשתמש ב - `chuya` בשביל להחזיר `Promise` מהפונקציה:

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>Document</title>
</head>

<body>
    <script>
        async function asyncFunction() {
            return "value";
        }

        asyncFunction().then(alert);
    </script>
</body>
</html>
```

בנוסף יחד עם `async` נוספה לשפה המילה השמורה `await` . השימוש ב`await` יכול להיות רק בתוך פונקציה שהוגדרה בתוך `async`, והוא משמשת על מנת לגרום לכך "להמתין" לסיום של פעולה אסינכרונית כלשהי.

לדוגמה:

```

<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>Document</title>
</head>

<body>
    <script>
        async function asyncFunction() {
            let promise = new Promise((resolve, reject) => {
                setTimeout(() => resolve("value"), 1000)
            })
            let result = await promise;

            alert(result);
        }

        asyncFunction();
    </script>
</body>
</html>

```

בקוד שראינו, אפשר לראות שהמשתנה `result` " ממתיין" ל `Promise`. הקוד נכתב בצורה המדימה קוד סינכרוני, אבל כמובן המשתנה יקבל את הערך שלו בצורה אסינכרונית לאחר مليוי ה `Promise` .

הכתיבה הנכונה וה `best practice` בשימוש ב `async` ו `await` , הוא "לעטוף" את `await` בתוך בלוק של `try catch` . פעולה אסינכרונית היא פעולה שמחכה להסתיימים מתישיה בעתיד, ולכן קיימים גם הסיכוי שפעולה זו תיכשל. בשבייל "להגן" על הקוד שלנו ולהיות ערוכים לסוגים כלשהם של כשלונות, וכן בשבייל לנחל את השגיאות בצורה יותר נוחה. נרצה להשתמש ב `try catch` .

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>Document</title>
</head>

<body>
    <script>
        async function asyncFunction() {

            try {
                let response = await fetch('http://domain.com/users');
                let user = await response.json();
            }
            catch(error) {
                console.error(error);
            }
        }

        asyncFunction();
    </script>

</body>
</html>
```

7. מחלקות הורשה ו-prototype

Constructor function .7.1 |

אפשרי ליצור מעין "מחלקה" ממנה ניתן ליצור אובייקטים ע"י האופרטור `new`. את הפונקציה נוצר כמו כל פונקציה רגילה – וניתן לה שם שמייצג את הטיפוס של האובייקטים שנוצר ממנה בהמשך.

בתוך הפונקציה זו – נרשם כל מאפיין או פונקציה שנרצה לאפשר לאובייקט שיופיע – על ידי המקדם `this`.

כאשר נקרא לפונקציה על ידי האופרטור `new` יוצר אובייקט חדש – והמליה `this` תהייה `context` של אותו אובייקט.

לדוגמה – ניצור constructor function של `Cat` שמכיל שני מאפיינים: `color`, `age` וניצור אובייקט ע"י הקראיה `()`.
`new Cat()`

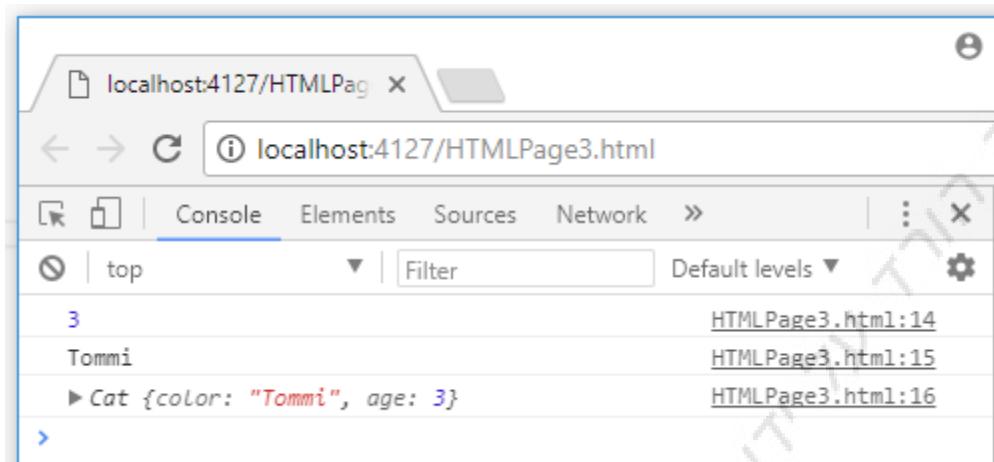
```
<!DOCTYPE html>

<html lang="en" xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta charset="utf-8" />
    <title></title>
<script>
    function Cat(color, age) {
        this.color = color;
        this.age = age;
    }

    var c = new Cat("Tommi", 3);
    console.log(c.age);
    console.log(c.color);
    console.log(c);

</script>
</head>
<body>

</body>
</html>
```



הוספת פונקציות ל - Constructor Function

1. נוכל להוסיף פונקציות שיתאפשרו להפעלה על ידי האובייקטים שייווצרו, בשני דרכים שונים:
באותה הדרך בה הוספנו מאפיינים – כמו למשל: ניצור משתנה עם הקידמת this ולתוכו נאחסן פונקציה.
2. ע"י פניה מוצלח constructor function של המודול – בדרך זו הפונקציה לא תוווצר עבור כל אובייקט – וילך בצורה זו ייחס מקומם בזיכרון.

שימוש לב: בשתי הדריכים – השימוש במילה this בתוך הפונקציה, יתיחס לאובייקט הספציפי דרכו הופעלה הפונקציה.



להלן דוגמה המօיפה לCat שיצרנו בדוגמה הקודמת, פונקציה בשם drinkMilk:

```

<!DOCTYPE html>

<html lang="en" xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta charset="utf-8" />
    <title></title>
<script>
    function Cat(color, age) {
        this.color = color;
        this.age = age;

        /*
        -----first way:
        Will duplicate the function code to each future object:
        this.drinkMilk = function () { };
        */
    }

```

```

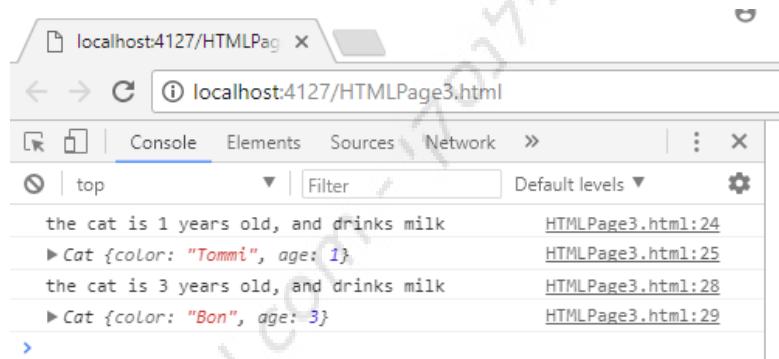
//----second way:
// Won't duplicate function code to any future object, but will be placed at the
prototype once.
Cat.prototype.drinkMilk = function () {
    return `the cat is ${this.age} years old, and drinks milk`;
};

var c1 = new Cat("Tommi", 1);
console.log(c1.drinkMilk());
console.log(c1);

var c2 = new Cat("Bon", 3);
console.log(c2.drinkMilk());
console.log(c2);

</script>
</head>
<body>

</body>
</html>
```



יצירת מאפיינים ופונקציות סטטיות

על ידי גישה ל-prototype של הפונקציה – נוכל להוסיף לה משתנים ופונקציות סטטיות שייהו נגישים בקורס ללא צורך ביצירת אובייקט של אותה הפונקציה על ידי new:

```
<!DOCTYPE html>

<html lang="en" xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta charset="utf-8" />
    <title></title>
<script>

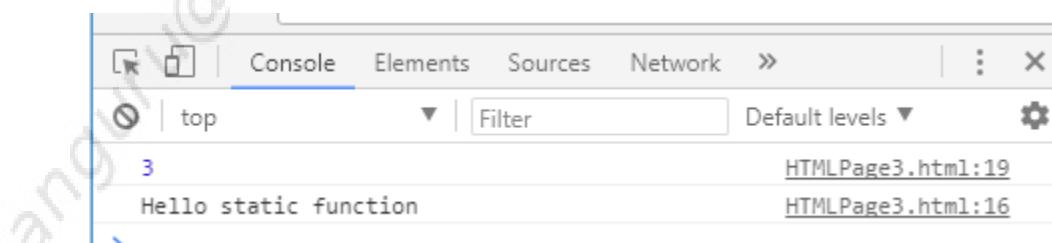
    function Cat(color, age) {
        this.color = color;
        this.age = age;
    }

    Cat.minWeight = 3; // Static variable.
    Cat.sayHello = function () { // Static function.
        console.log("Hello static function");
    };

    console.log(Cat.minWeight);
    Cat.sayHello();

</script>
</head>
<body>

</body>
</html>
```



הוספת משתנה ל private Constructor Function

בכדי למש את עקרון encapsulation המורה כי על מחלקות לשמור על כלליים, ולבצע בדיקות ואלידציה עבור משתנים באופן פנימי, נוכל להוסיף ל-constructor function משתנים פרטיים, באופן הבא:

```
<!DOCTYPE html>

<html lang="en" xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta charset="utf-8" />
    <title></title>
<script>
    function Cat(color, age) {
        this.color = color;

        //private member - will be created for each object, and will continue to be
        alive in the memory for that object
        //because private variables are created for a function and exists for that
        function as long as the function exists.
        //They do not destroyed when the function completes.
        var _age = 0;

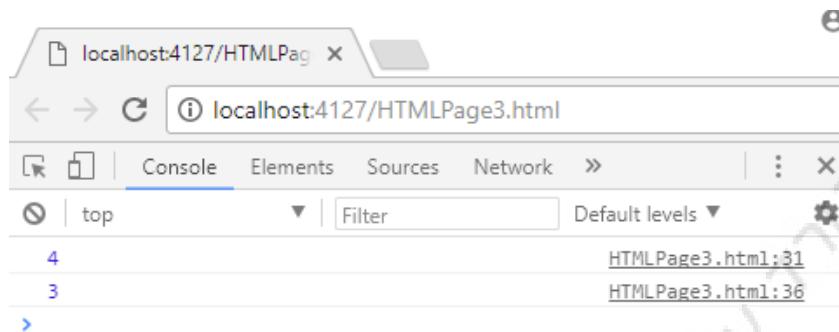
        this.getAge = function () {
            return _age;
        }
        this.setAge = function (val) {
            if (val > 0) {
                _age = val;
            }
        }
        this.setAge(age);
    }

    // Setting legal values:
    var c1 = new Cat("Tommi", 1);
    c1.setAge(4);
    console.log(c1.getAge());

    // Setting illegal values:
    var c2 = new Cat("Bon", 3);
    c2.setAge(-4);
    console.log(c2.getAge());


</script>
</head>
<body>

</body>
</html>
```



Inheritance

אם נרצה ליצור Object של Animal וlhagdir shet Cat יירש מ, נוכל לעשות זאת
בצורה הבאה:

```
<!DOCTYPE html>

<html lang="en" xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta charset="utf-8" />
    <title></title>
<script>

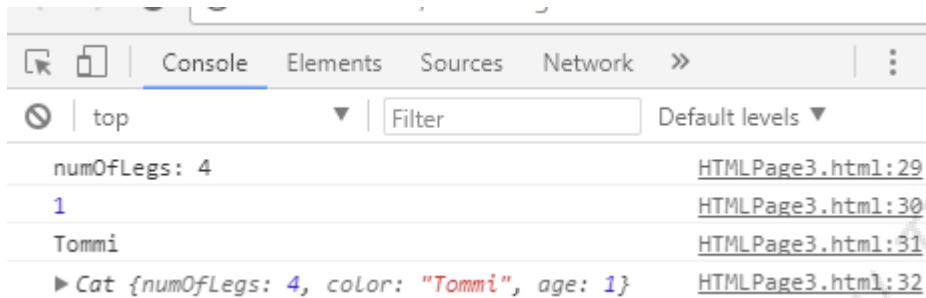
    function Animal(numOfLegs) {
        this.numOfLegs = numOfLegs;
    }

    Animal.prototype.getNumOfLegs = function () {
        return `numOfLegs: ${this.numOfLegs}`;
    };

    function Cat(numOfLegs,color, age) {
        Animal.call(this, numOfLegs);
        this.color = color;
        this.age = age;
    }
    Cat.prototype = Object.create(Animal.prototype);

    var c1 = new Cat(4,"Tommi", 1);
    console.log(c1.getNumOfLegs());
    console.log(c1.age);
    console.log(c1.color);
    console.log(c1);
</script>
</head>
<body>

</body>
</html>
```



Polymorphism

בכדי למש את עיקנון הפולימורפיזם, שהינו אחד מאבי היסוד בתכנות מונחה עצמים, נוכל ליצור function constructor של Animal ולהגדיר שאל Cat יירוש מ, כאשר לשנייהם יש את הפונקציה getInfo וכן אחד מבצע אותה במימוש המתאים לו, תוך שימוש במחלקה הבסיס ממנה ירש. נוכל לעשות זאת בצורה הבאה:

```

<!DOCTYPE html>

<html lang="en" xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta charset="utf-8" />
    <title></title>
<script>

    function Animal(numOfLegs) {
        this.numOfLegs = numOfLegs;
    }

    Animal.prototype.getInfo = function () {
        return `numOfLegs: ${this.numOfLegs}`;
    };


    function Cat(numOfLegs,color, age) {
        Animal.call(this, numOfLegs);
        this.color = color;
        this.age = age;
    }
    Cat.prototype = Object.create(Animal.prototype);

    Cat.prototype.getInfo = function () {

        // Animal.prototype.getInfo.call(this) equals to base.ToString() in C# or
super.toString() in Java
        return Animal.prototype.getInfo.call(this) + `, color: ${this.color}, age:
${this.age}`;
    };

```

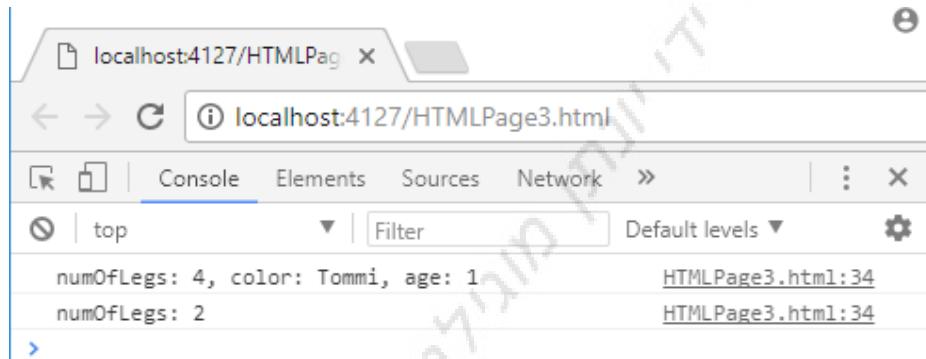
```

var arr = [new Cat(4, "Tommi", 1), new Animal(2)];
for(let i of arr) {
    console.log(i.getInfo());
}

</script>
</head>
<body>

</body>
</html>

```



Functions as Namespaces

למשתנים המוגדרים בתחום פונקציה יש טווח הכרה לאורך הפונקציה (כולל בתחום פונקציות מקומיות), אבל הם לא קיימים מחוץ לפונקציה. המשתנים המצוירים מחוץ לפונקציה הם משתנים גלובליים והם גלויים לאורך כל תוכנית.. אולם נוכל בכל זאת ליצור משתנים מוסתרים בתחום בלוק קוד, על ידי הגדרת פונקציה פשוטה שתשתמש כ-namespace זמני שבו ניתן להגדיר משתנים מביי להוסיפם ל- global namespace

כנראה, לדוגמה, שיש לנו מודול של קוד JavaScript וברצוניו להשתמש בו במספר תוכניות JavaScript שונות, ונניח שקיים זה מגדר מושגים לאחסון תוכאות הביניים של החישוב שלו. הבעיה היא שפנוי שהמודול הזה ישמש בתוכניות רבות ושונות, איננו יודעים אם המשתנים שהוא יוצר יתנגשו עם משתנים המשמשים את התוכניות המיויבות. הפתרון, כמובן, הוא לשים את הקוד בתחום פונקציה ולאחר מכן להפעיל את הפונקציה. בדרך זו, משתנים שהיו גלובליים הופכים להיות local :

```
function mymodule() {
    /*Module code goes here.

    Any variables used by the module are local to this function
    instead of cluttering up the global namespace.*/
}

mymodule(); // But don't forget to invoke the function
```

קוד זה מגדיר רק משתנה גלובלי יחיד: שם הפונקציה "mymodule". אולם אפשר לחסוך גם את ההגדה של המשתנה היחיד זהה ע"י הגדרת פונקציה אונומית בביטוי אחד:

```
(function () {      // mymodule function rewritten as an unnamed expression
    // Module code goes here.
}()); // end the function literal and invoke it now.
```

class . 7.2 |

לפני ES6, יצירתי `class` הייתה עניין מסוים. ב-ES6 ניתן ליצור `class` באמצעות ה- keyword החדש `.class`

מחלקות יכולות להיכלל בקוד או על ידי הכרזה או על ידי שימוש בביטויים השמאליים:

- Declaring a Class

```
class Class_name {  
}
```

- Class Expressions

```
var var_name = new Class_name {  
}
```

הגדרת מחלקה יכולה לכלול את הפריטים הבאים:

- **Constructor** - מילה שמורה היוצרת פונקציה שאחראית על הקצתה זיכרון עבור אובייקטים של הכיתה.

- **methods** - methods מייצגות פעולות שאובייקט יכול לבצע. הם נכתבים ללא הקידומת של **function**.
המילה השמורה **members** מרכיבים אלה ביחד ביחד **data members** של המחלקה.

הערה - גוף מחלקה יכול להכיל רק **methods**, אך לא **data properties**.



לדוגמה, הגדרת מחלקה:

```
class Circle {
    constructor(height, width) {
        this.height = height;
        this.width = width;
    }
}
```

לדוגמה, הגדרת מחלקה ע"י ביטוי:

```
var Circle = class {
    constructor(height, width) {
        this.height = height;
        this.width = width;
    }
}
```

קטע הקוד שלמעלה מיצג ביטוי מחלקה ללא שם. ביטוי השמה של מחלקה גם עם שם:

```
var Circle = class Circle {
    constructor(height, width) {
        this.height = height;
        this.width = width;
    }
}
```

הערה – בנויגוד למשתנים ולפונקציות, מחלקות לא מבצעות hoisting – ולא מוכרכות מעיל השורות בהן הוגדרו.



יצירת אובייקטים

כדי ליצור מופע של המחלקה, יש להשתמש ב- **new** ואחריו לציין את שם המחלקה. להלן התוצאות:

```
var object_name = new class_name([arguments])
```

לדוגמה, הגדרת מחלקה ויצירת מופע שלה:

```
<!DOCTYPE html>

<html lang="en" xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta charset="utf-8" />
    <title></title>
    <script>

        var Circle = class Circle {
            constructor(height, width) {
                this.height = height;
                this.width = width;
            }
        }

        var c = new Circle(30, 40);
    </script>
</head>
<body>

</body>
</html>
```

גישה לפונקציות

ניתן לארת למאפיינים ולפונקציות של המחלקה באמצעות שם האובייקט בתוספת סימן 'נקודה' וזר פניה לפונקציה. לדוגמה:

```
<!DOCTYPE html>

<html lang="en" xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta charset="utf-8" />
    <title></title>
    <script>

        var Circle = class Circle {
            constructor(height, width) {
                this.height = height;
                this.width = width;
            }

            print() {
                console.log(` ${this.height} ${this.width}`);
            }
        }

        var c = new Circle(30, 40);
        c.print();
```

```

        </script>
</head>
<body>

</body>
</html>
```

Static Keyword

ניתן להחיל את המילה השמורה static על פונקציות במחלקה. מוגדרות סטטיות נגישות דרך שם המחלקה. לדוגמה:

```

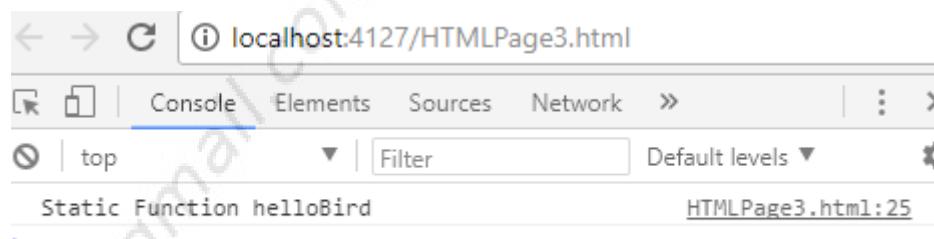
<!DOCTYPE html>

<html lang="en" xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta charset="utf-8" />
    <title></title>
    <script>

        class Bird {
            static helloBird() {
                console.log("Static Function helloBird")
            }
        }
        Bird.helloBird() //invoke the static method

    </script>
</head>
<body>

</body>
</html>
```



instanceof operator

האופרטור instanceof מחזיר true אם האובייקט שיר לסוג שצוין. לדוגמה:

```
<!DOCTYPE html>

<html lang="en" xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta charset="utf-8" />
    <title></title>
    <script>

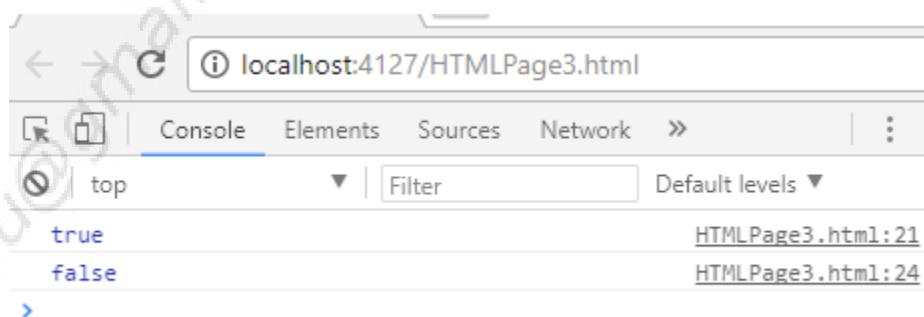
        var Circle = class Circle {
            constructor(height, width) {
                this.height = height;
                this.width = width;
            }

            print() {
                console.log(` ${this.height} ${this.width}`);
            }
        }

        var c = new Circle(30, 40);
        console.log(c instanceof Circle);

        var obj = {};
        console.log(obj instanceof Circle);
    </script>
</head>
<body>

</body>
</html>
```



Class Inheritance

ES6 תומכת במושג הירושה. ירשה היא היכולת של תוכנית ליצור תכניות של ישויות חדשות מtabניות ישות קיימת - המחלקה הבסיסית שמשמשת מחלקות חדשות יותר נקראת מחלקה האב. ומחלקות החדשות שיורשות ממנה מכונות נגזרות.

מחלקה אחרת יורשת ממחלקה אחרת באמצעות המילה השמורה `extends`. נגזרות יורשת את כל המאפיינים והשיטות, למעט בנאי מחלקה האב.

להלן התchapir ליצירת מחלקה יורשת:

```
class child_class_name extends parent_class_name
```

דוגמה מלאה:

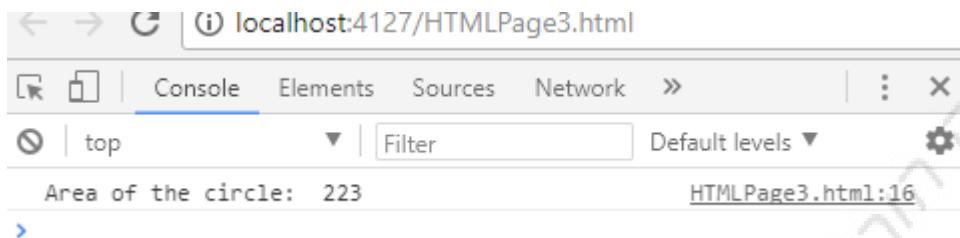
```
<!DOCTYPE html>

<html lang="en" xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta charset="utf-8" />
    <title></title>
    <script>

        class Shape {
            constructor(a) {
                this.Area = a
            }
        }
        class Circle extends Shape {
            disp() {
                console.log("Area of the circle: " + this.Area)
            }
        }
        var obj = new Circle(223);
        obj.disp()

    </script>
</head>
<body>

</body>
</html>
```



ניתן לסווג את הירושה כ -

- ירושה יחידה - כל מחלוקת יכולה, לפחות, לפחות, להאריך מחלוקת בסיס אחת
- ירושה מרובה - מחלוקת יכולה לרשות בירושה מכמה מחלוקת בסיס - יכולות זו לא מתאפשרת בסיס 6ES.
- ירושה **Multi-level** - לדוגמה:

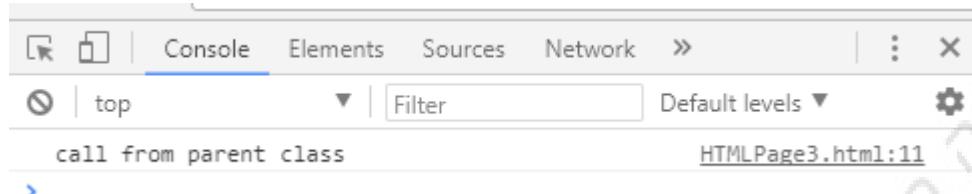
```
<!DOCTYPE html>

<html lang="en" xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta charset="utf-8" />
    <title></title>
    <script>

        class A {
            test() {
                console.log("call from parent class")
            }
        }
        class B extends A {}
        class C extends B {}

        //C inherits from A and B
        var obj = new C();
        obj.test()

    </script>
</head>
<body>
</body>
</html>
```



Method Overriding

הוא מצב שבו הנגזרת מגדרה מחדש את אותה שיטה שכבר הוגדרה בבסיס. הדוגמה הבאה ממחישה את הדבר:

```
<!DOCTYPE html>

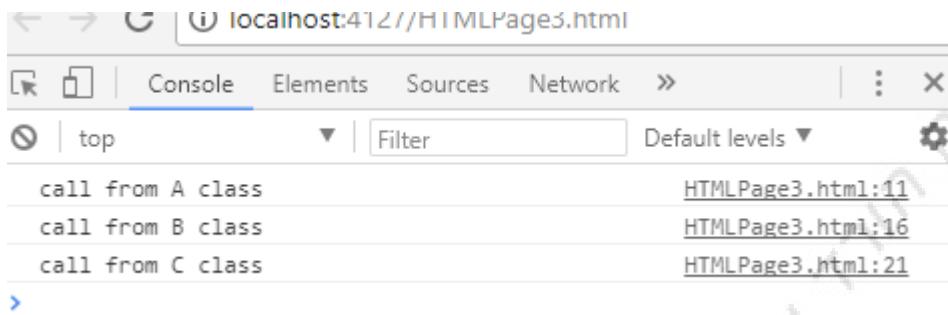
<html lang="en" xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta charset="utf-8" />
    <title></title>
    <script>

        class A {
            test() {
                console.log("call from A class")
            }
        }
        class B extends A {
            test() {
                console.log("call from B class")
            }
        }
        class C extends B {
            test() {
                console.log("call from C class")
            }
        }

        var obj1 = new A();
        obj1.test();
        var obj2 = new B();
        obj2.test();
        var obj3 = new C();
        obj3.test();

    </script>
</head>
<body>

</body>
</html>
```



Super Keyword

6 אפשר לנגרת, להפעיל את המethod של מחלקת הבסיס זה מושג באמצעות המילה השמורה super המשמשת להפניה להורה הישיר של המחלקה. לדוגמה:

```
<!DOCTYPE html>

<html lang="en" xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta charset="utf-8" />
    <title></title>
    <script>

        class A {
            test() {
                console.log("call from A class");
            }
        }
        class B extends A {
            test() {
                super.test();
                console.log("call from B class");
            }
        }
        class C extends B {
            test() {
                super.test();
                console.log("call from C class");
            }
        }

        var obj1 = new A();
        console.log("-----obj1-----");
        obj1.test();

        var obj2 = new B();
        console.log("-----obj2-----");
        obj2.test();

        var obj3 = new C();
        console.log("-----obj3-----");
        obj3.test();

    </script>
</head>
<body>
</body>
</html>
```

```
</script>
</head>
<body>

</body>
</html>
```

The screenshot shows a browser developer tools window with the 'Console' tab selected. The output area displays a series of log messages:

```
-----obj1-----
call from A class
-----obj2-----
call from A class
call from B class
-----obj3-----
call from A class
call from B class
call from C class
```

Each message is followed by its file name and line number: `HTMLPage3.html:28`, `HTMLPage3.html:11`, `HTMLPage3.html:32`, `HTMLPage3.html:11`, `HTMLPage3.html:17`, `HTMLPage3.html:36`, `HTMLPage3.html:11`, `HTMLPage3.html:17`, and `HTMLPage3.html:23`.

7.3. תרגילים



constructor function & inheritance

תרגיל 1

1. צרו Shape Constructor Function בשם Shape המתארת צורה כללית ומכליה:

- x – מיקום הצורה על ציר x.
- y – מיקום הצורה על ציר y.
- color – צבע הצורה.

הוסף לו ל-Shape פונקציה המחזיר את מרחק הצורה מראשית הצירים.

ຄלומר מרחק הנקודה (y,x) מראשית הצירים.

נוסחה: שורש של (x בריבוע + y בריבוע).

1. בצעו דרישת `toString` והחזירו ממנה מחזורת המכיל את פרטי הצורה בפורמט הבא:

X = _____, Y = _____, Color = _____

תרגיל 2

1. צרו `Shape` בשם `Circle` המתארת עיגול, ע"י הורשת מחלקת `Shape`-`Constructor Function`. על העיגול להכיל בנוסף ל-`x`, ל-`y` ול-`color` שהגינו מ-`Shape`, גם `radius` – רדיוס העיגול.

2. בצעו דרישת `toString` פונקציית `toString` כך שהפעם היא תחזיר את פרטי העיגול בפורמט הבא:

X = _____, Y = _____, Color = _____, Radius = _____

3. הוסיףו ל-`Circle` מאפיין סטטי בשם `PI` השווה ל-`3.14`.

4. הוסיףו ל-`Circle` פונקציה נוספת בשם `getArea` המחזיר את שטח הפנים של העיגול.

5. הוסיףו ל-`Circle` פונקציה נוספת בשם `getPerimeter` המחזיר את היקף העיגול.

6. השתמשו ב-`Circle` שבנוים ע"י ביצוע הפעולות הבאות:

- צרו אובייקט `Circle` בעל `x`, `y`, `color` ו-`radius` כלשהם.
- הציגו את הערך המוחזר מה-`toString`.
- הציגו את המרחק מראשית הצירים.
- הציגו את שטח הפנים של העיגול.
- הציגו את היקף העיגול.
- הציגו את `PI`.

תרגיל 3

צור את המחלקות הבאות:

מחשב

מכל את המאפיינים:

- זיכרון מעבד (4-16 GB)
- זיכרון DISK (200-3000 GB)

- דגם מעבד
- מחיר (000-20000)
- שנות אחריות (5-0)

מכיל את הפונקציות:

- רכישת ציוד נלווה - מדפסה הצעה לרכישת אוזניות
- הדפסת פרטי המחלקה - print

מחשב נייח (ירוש מחשב)

מכיל את המאפיינים:

- האם הUBLICOVER ALCHOTI (בוליאני)
- גודל מסך מחשב נייח (18-11)

מכיל את הפונקציות:

- רכישת ציוד נלווה - מציגה ללקוח הצעה לרכישת שולחן מחשב
- הדפסת פרטי המחלקה - print

מחשב נייד (ירוש מחשב)

מכיל את המאפיינים:

- מספר שעות הטענה (9-1)
- אחוז סוללה (0-100)
- האם מסך מגע (בוליאני)

מכיל את הפונקציות:

- רכישת ציוד נלווה - מציגה ללקוח הצעה לרכישת תיק למחשב נייד + קרייה לפונקציית הבסיס
- הטענת המחשב הנייד - פונקציה המציגה הודעה שסוללת המחשב הוטענה בהצלחה
- הדפסת פרטי המחלקה - print

1. צור פונקציה בשם **executeActions** המתקבלת משתנה ומבצעת את כל הפונקציות שיש לאוותנו
אובייקט נגזרת
2. צור מערך באורך 10 תאים
3. אתחל כל תא בעל אינדקס זוגי למחשב נייד, וכל תא בעל אינדקס אי-זוגי למחשב נייח
4. שלח כל תא במערך לפונקציה **executeActions**

תרגיל בנושא function as namespace



הוסףו לפרויקט 2 קבצי JavaScript בעלי השמות הבאים:

globals.js

site.js

צרו ב-index.html קישור לסק립טים הללו:

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

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

(יש לשים לב שה קישור ל-.js globals ומצא מעל הקישור ל-.js site)

בקובץ ה-.js globals בצעו:

1. Self-Invoked Function העוטפת את כל תוכן הקובץ

2. "use strict"

3. Namespace globals בשם window השיר ל-

4. פונקציה בשם getCurrentTime המחזירה את השעה הנוכחיית (מחזירה ע"י return, לא מציגה ע"י alert)

בקובץ ה-.js site בצעו:

1. Self-Invoked Function העוטפת את כל תוכן הקובץ

2. "use strict"

3. קראו לפונקציה getCurrentTime שנמצאת ב-Namespace site שיצרתם, והציגו את הערך המוחזר ממנה ע"י alert

הריצו את האתר וביראו שכאן מוצגת השעה הנוכחית.