Advanced JavaScript

27766-4578-02

JOHN BRYCE
תלמד הי-טק. זה עובד!
a matrix company

Advanced JavaScript

Including es5+es6

תוכן עניינים

1. מבוא

JavaScript ES פותח על ידי Brendan Eich, מפתח ב- Netscape Communications Corporation, בשנת 1995.
זוהי שפת scripting המבוצעת על ידי הדפדפן, כלומר, בצד הלקוח. ומשמשת בשילוב עם HTML לפיתוח דפי אינטרנט דינמיים.

ECMAScript היא ספסיפיקציה לשפת סקריפט. במקור ECMA הן ראשי תיבות של European Computer Manufacturers Association - התאחדות אירופאית ליצרני מחשבים. ארגון ללא מטרות רווח שמטרתו היא לקבוע סטנדרטים לתקשורת. הסטנדרט המוכר ביותר של הארגון הוא. ECMAScript
שפת JavaScript בנויה על בסיס הסטנדרטים של ECMAScript – כלומר ECMAScript הן ההוראות שלפיו JavaScript צריכה להתיישר.

ECMAScript 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.

ECMAScript 5 ו- ECMAScript6 – 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

שימו לב, עדיין לא כל הדפדפנים הטמיעו את היכולות להריץ את הקוד בתחביר החדש, אולם מכיוון שהשימוש בה נפוץ מאד, ומאפשר ליצור קודJavaScript קריא יותר וקל לתחזוק, רבים מהמתכנתים מעדיפים בכל זאת להשתמש בגרסאות החדשות של- ECMAScript, אפשר להשתמש בנוסף גם ב- Babel שלוקח קוד שלJavaScript שכתוב לפי התקן החדש ביותר ועושה לו טרנספיילינג (המרה של קוד אחד בקוד אחר באותה רמת אבסטרקציה) ומשלב בתוכו שכתוב של הקוד כך שיתאים לתקנים ישנים יותר.

2. סוגי משתנים - Data types

2.1. טיפוס משתנה דינמי

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

בעקבות זאת, JavaScript מאפשרת לאותו משתנה לקבל ערכים מטיפוסים שונים. 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 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. סוגי 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 המייצגים תאריכים. המחלקה RegExp מגדירה אובייקטים המייצגים ביטויים רגולריים (pattern-matching). והמחלקה Error המגדירה אובייקטים המייצגים שגיאות זמן ריצה שעלולים להתרחש בתוכנת JavaScript.
- אובייקט הוא אוסף של מאפיינים שבהם לכל מאפיין יש שם וערך (או ערך פרימיטיבי כגון מספר, או אובייקט).
- אובייקט JavaScript רגיל, הוא unordered collection של מפתחות וערכים.
- JavaScript מגדירה גם סוג ordered collection, הידוע כמערך (Array) המייצג אוסף מסודר של ערכים ממוספרים ע"י אינדקס מספרי ולא ע"י מפתח מחרוזתי.
- פונקציה מוגדרת ב- JavaScript כטיפוס של אובייקט.

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

mutable and immutable types:

- **mutable** - ערך של משתנה מסוג mutable יכול להשתנות
קבוצת האובייקטים הם mutable
- **immutable** - number, boolean, null, undefined string וכו', אינם ניתנים לשינוי.

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

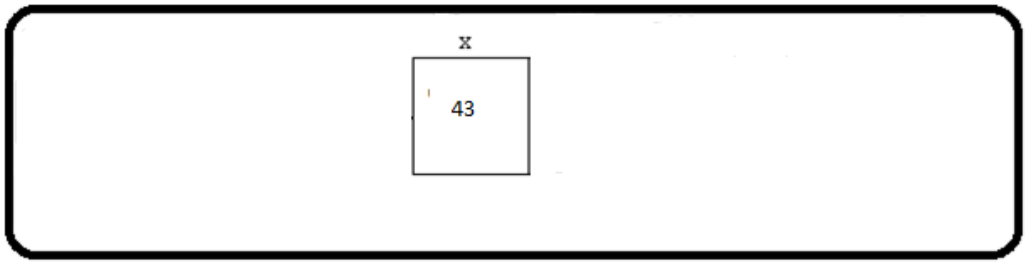
2.3. Ref type VS value type

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

לדוגמא, כאשר ניצור את ההגדרה הבאה:

```
var x=43;
```

בזיכרון יוצר המצב הבא:



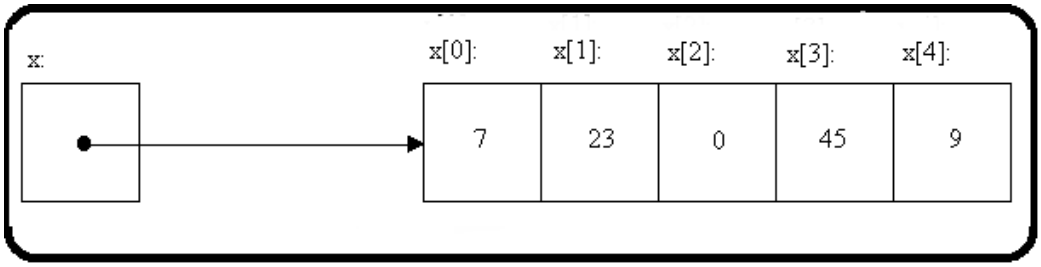
אזור זיכרון

- המשתנים מסוג **reference type** הם משתנים מטיפוס object type, והמשתנה מכיל הפניה לשטח שבו האובייקט נוצר.

לדוגמה, עבור ההגדרה:

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

נקבל בזיכרון את המפה הבאה:



אזור זיכרון

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

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

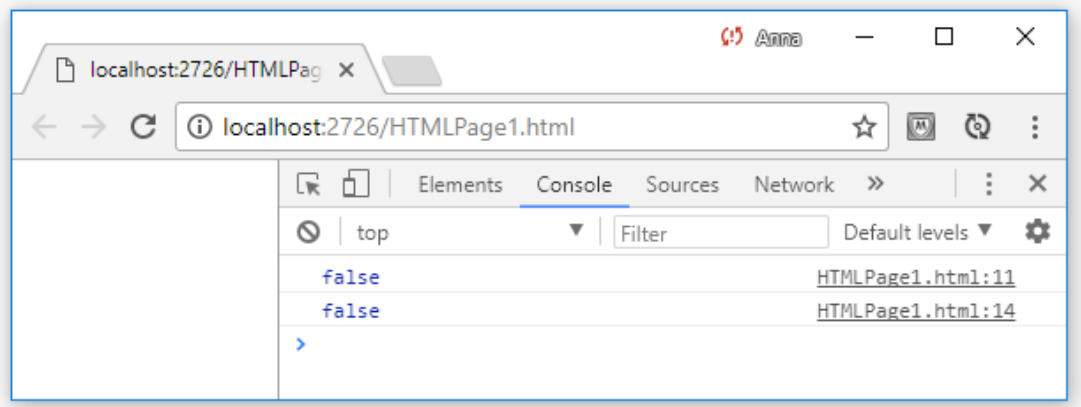
- במשתני value type – ייבדק האם שני המשתנים מכילים אותו ערך.

- במשתני reference type – ייבדק האם שני המשתנים מכילים הפניה לאותו אובייקט.

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>

</body>
</html>
```

כאשר נריץ את הקוד בדפדפן, נקבל את התוצאה הבאה:



1. בביצוע השמה של משתנה אחד לתוך משתנה אחר:
   - במשתנה מסוג value type – תבוצע העתקת הנתון שבתוך המשתנה
   - במשתנה מסוג reference type – תבוצע העתקת ההפניה אליה מצביע המשתנה

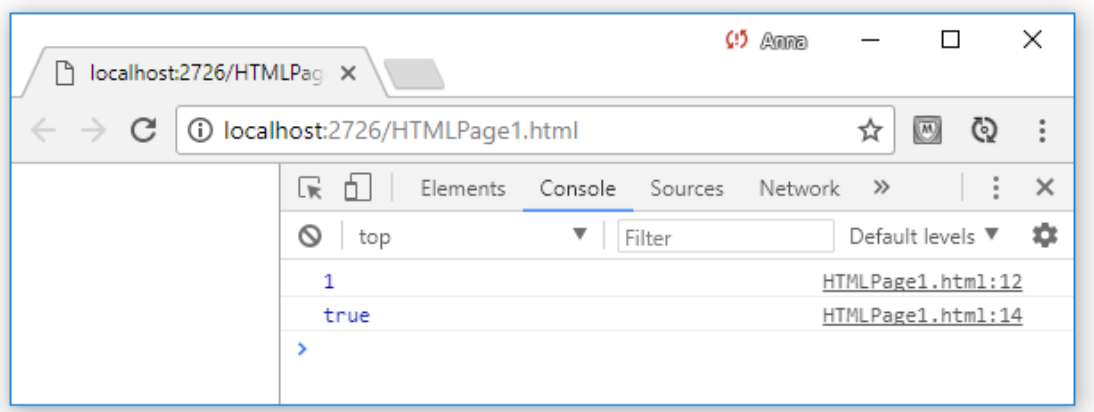
כאשר נבצע שינוי על העותק של משתנה פרימיטיבי, משתנה המקור לא יושפע.

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

לדוגמה, ניצור מערך בתוך משתנה a, נעתיק את תוכן a ל-b, ונראה שכל שינוי שביצענו 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:



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

1) העברת פרמטר **by value-** תתבצע כאשר נעביר משתנים פרימיטיביים.
קוד שקורא לפונקציה ושולח לה ערך של תוכן משתנה שתוכנו מועתק לתוך הפרמטר הלוקלי של הפונקציה.

2) העברת פרמטר **by reference**- תתבצע כאשר נעביר לפונקציה משתנה שהוא לא פרימיטיבי.
הקוד שקורא לפונקציה שולח לה ערך של כתובת למשתנה מסוים בזיכרון, והכתובת מועתקת לתוך הפרמטר הלוקלי של הפונקציה.

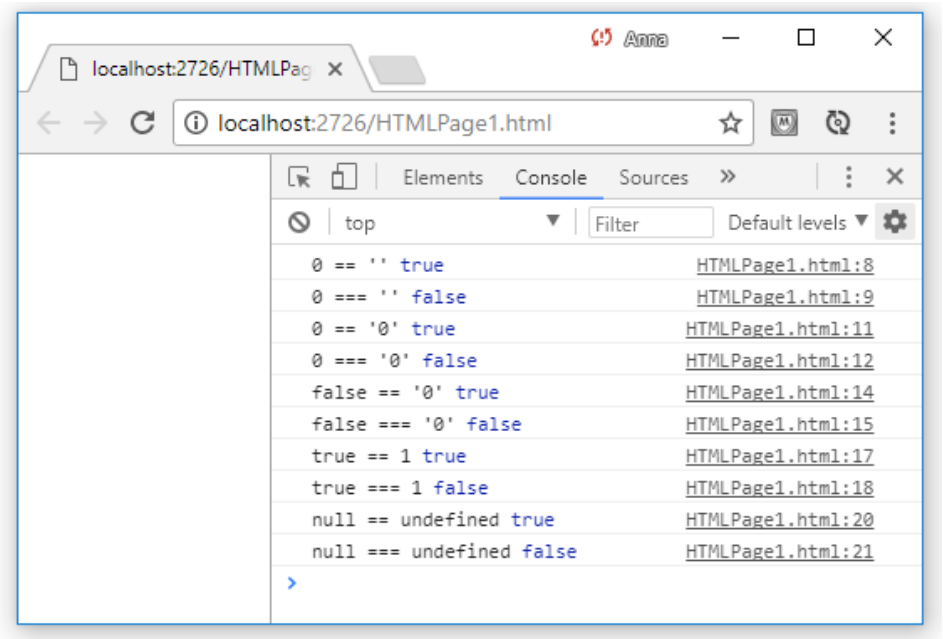
2.4. == VS ===

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

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 נריץ את הקוד בדפדפן, נקבל את התוצאה הבאה:



2.5. אופרטור typeof

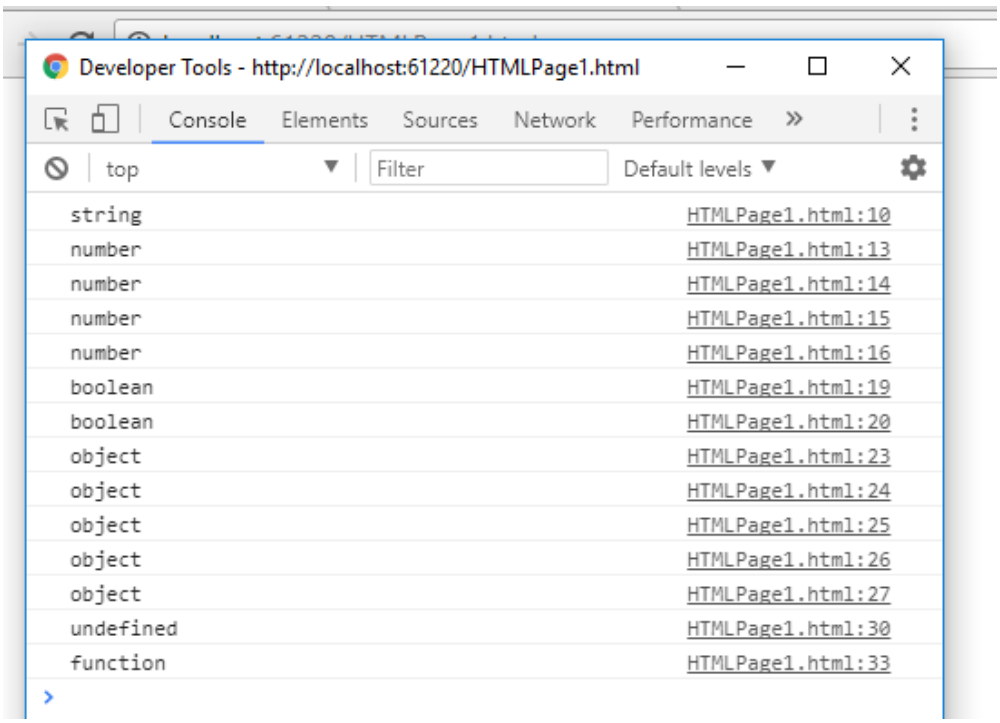
אופרטור typeof משמש כדי למצוא את סוג משתנה.

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()));
```

```
        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 נריץ את הקוד בדפדפן, נקבל את התוצאה הבאה:



ההבדל בין === ל-typeof

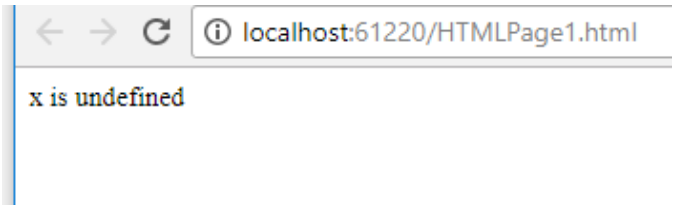
לעיתים קרובות נרצה לבדוק האם משתנה מסויים מכיל את הערך undefined.
ניתן לבצע את הבדיקה במספר דרכים:

- בדיקה באמצעות ===

לדוגמה, הקוד הבא:

```
<!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>
```

כאשר נריץ את הקוד בדפדפן, נקבל את התוצאה הבאה:



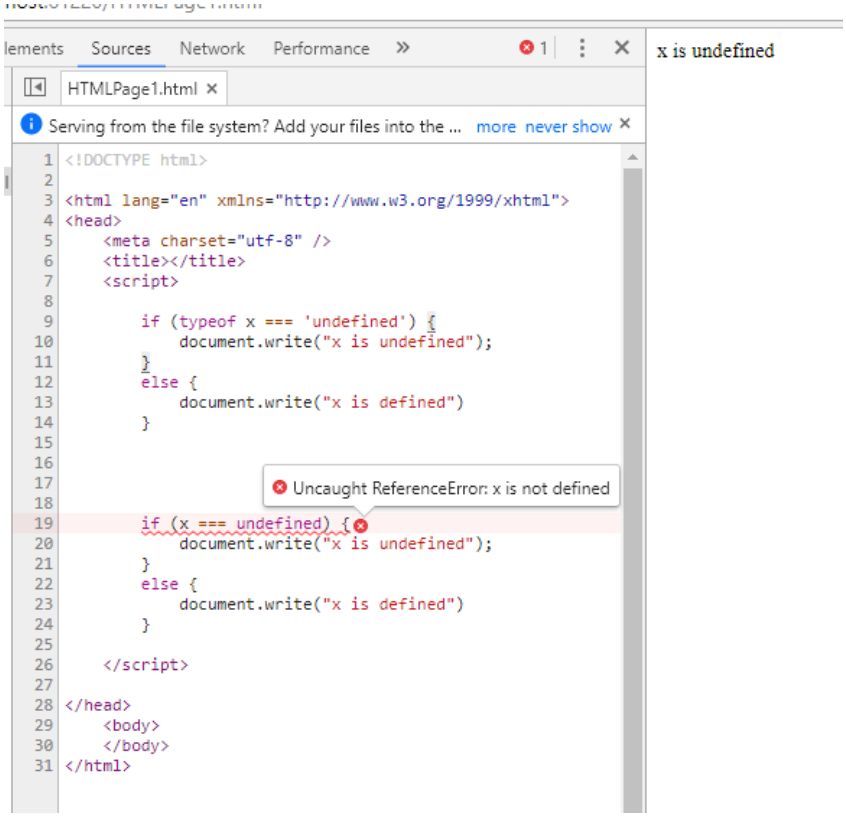
- בדיקה באמצעות **typeof**

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")
        }
```

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

    </script>

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

כאשר נריץ את הקוד בדפדפן, נקבל את התוצאה הבאה:



סיכום:

- typeof יכול גם לעבוד על משתנה udefined
- === יזרוק ReferenceError עבור שהמשתנה udefined

2.6. טיפוס number

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

number literals

- מספרים שלמים

לדוגמה: 123

- ערכים הקסדצימליים (בסיס 16)

hexadecimal literal מתחיל עם0**x** או **0X** ואחריו מספר הקסדצימלי.
ספרה הקסדצימלית מייצגת ערכים בטווח 0 עד 15, ויכולה להיות מיוצגת באמצאות אחת מהערכים הבאים:

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

לדוגמה: xff0

- **Floating-point literals** – יכולים 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

אופציונלי להוסיף סימן פלוס או מינוס, ואחריו מעריך שלם.
לדוגמה: 3-E1.473

2.7. טיפוס string

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

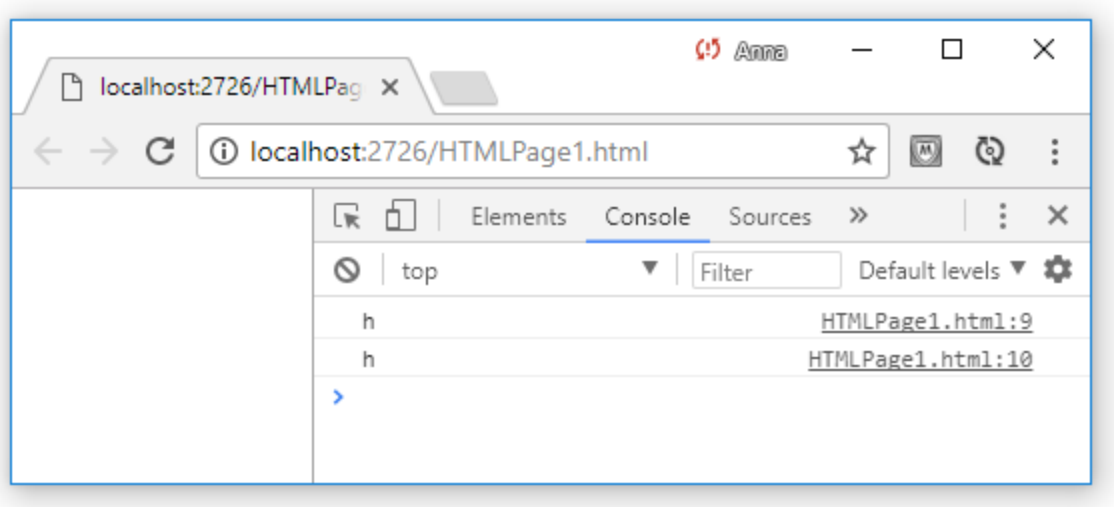
ל- JavaScript אין סוג מיוחד המייצג תו אחד של מחרוזת. וכדי לייצג ערך תו יחיד יש להשתמש במחרוזת בעלת אורך של תו אחד.

ב-ECMAScript ניתן לטפל במחרוזות כמו read-only arrays, ויש אפשרות לגשת לתווים בודדים של מחרוזת באמצעות סוגריים מרובעים במקום 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 בדפדפן, נקבל את התוצאה הבאה:

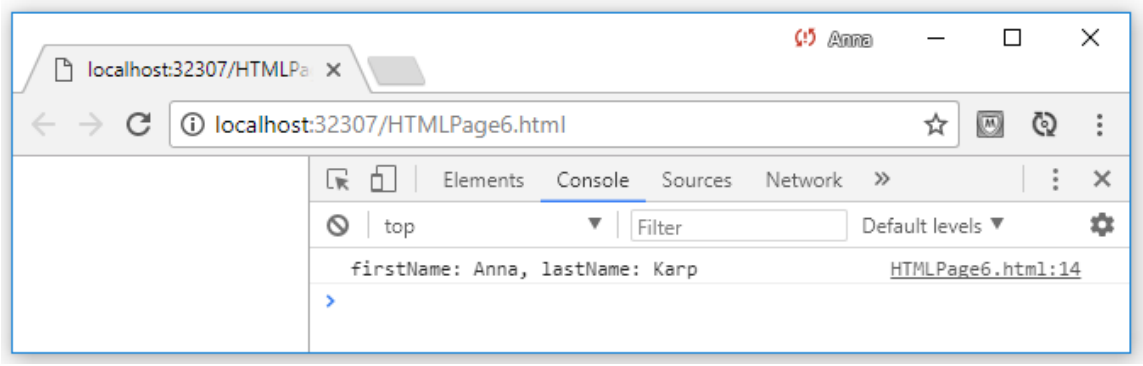


JavaScript Template String Literals

בכל פרויקט יהיו שימושים רבים באינטרפולציה 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>
```

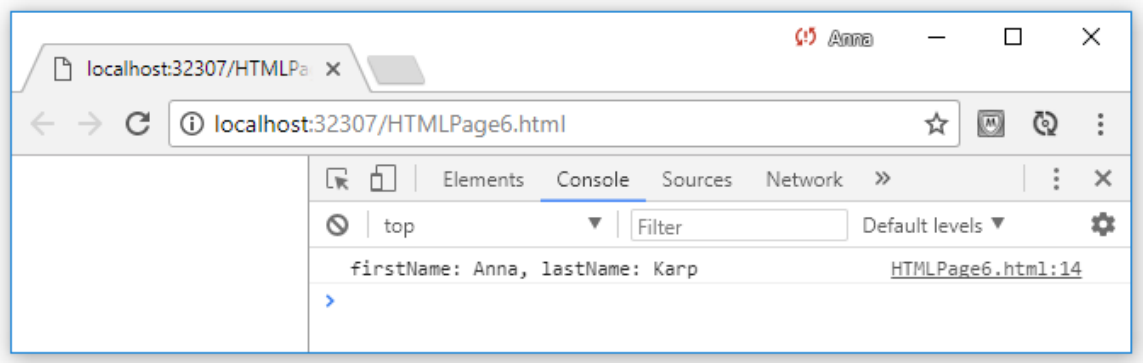
כאשר נריץ את הקוד בדפדפן, נקבל את התוצאה הבאה:



ECMAScript 2015 מביא פתרון הרבה 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>
```

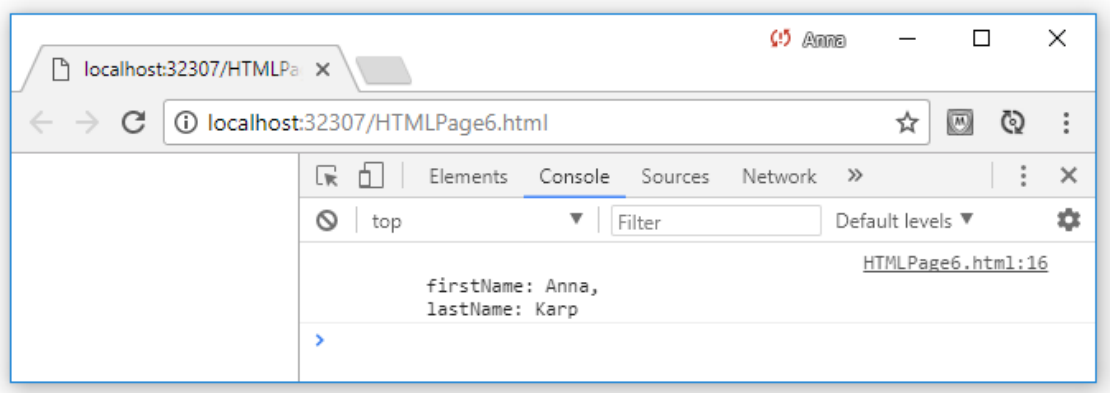
כאשר נריץ את הקוד בדפדפן, נקבל את התוצאה הבאה:



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>
```

כאשר נריץ את הקוד בדפדפן, 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 (ומערכים) מייצגים את הערך true.

2.9. טיפוס null ו - undefined

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

- הפעלת האופרטור typeof על null מחזירה את המחרוזת object

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

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

- הפעלת האופרטור typeof על undefined מחזירה את המחרוזת undefined

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

2.10. Wrapper Objects

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

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

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

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

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

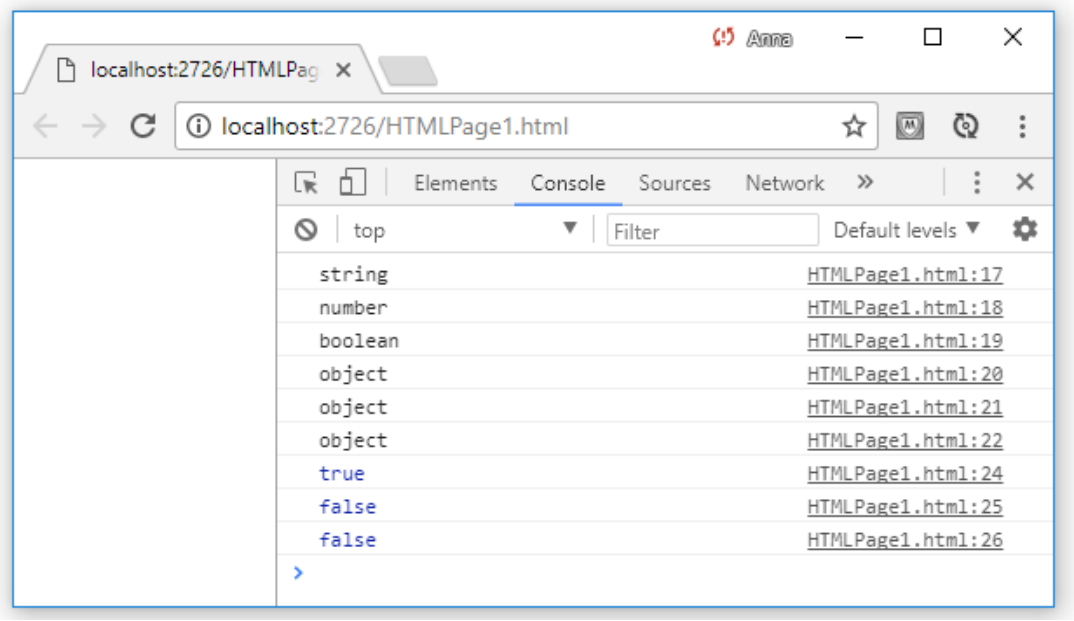
שים לב: ניתן ליצור אובייקטים של wrapper objects, על ידי שימוש בבנאים:

```
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>
```

כאשר נריץ את הקוד בדפדפן, נקבל את התוצאה הבאה:



האובייקטים S, N, ו- B בדוגמה לעיל יתנהגו בדרך כלל, בדיוק כמו הערכים s, n, b.
אולם האופרטור typeof והאופרטור === יראו את ההבדל בין primitive value ל wrapper object.

2.11. global object

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

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

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

2.12. Object and array initializers

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

```
var arr = [];       //empty array: no values inside brackets means no elements
var mat = [[1, 2, 3], [4, 5, 6], [7, 8, 9]];
```

אלמנטים המכילים undefined, יכולים להיכלל 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];
```

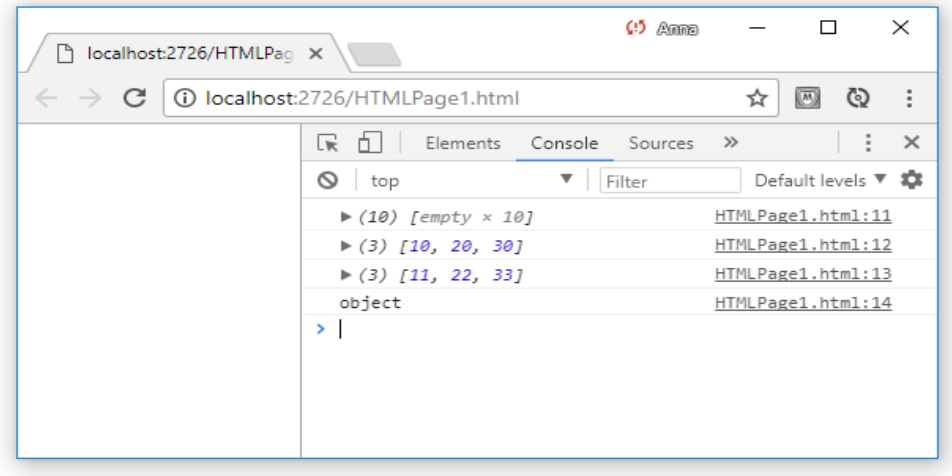
בנוסף, ניתן להגדיר מערך בדרך הבאה:

```
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>
```

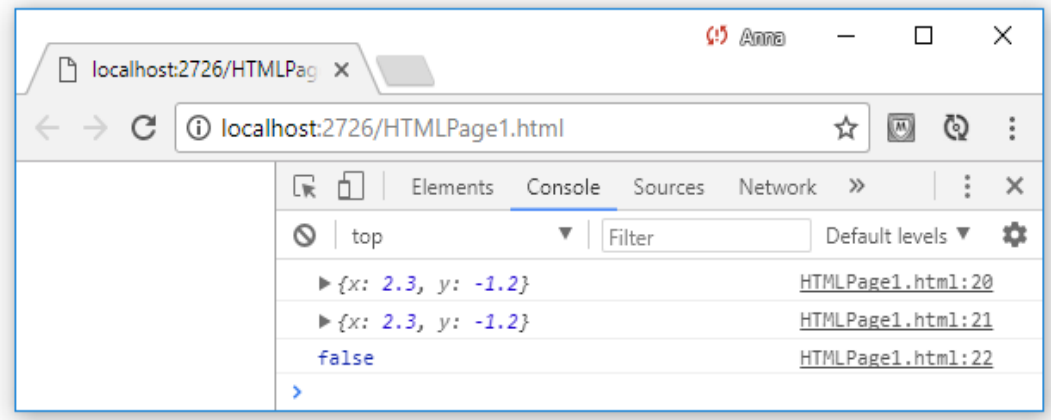
והתוצאה תהיה:



שימו לב: ניתן להוסיף למערכים איברים בצורה דינמית.

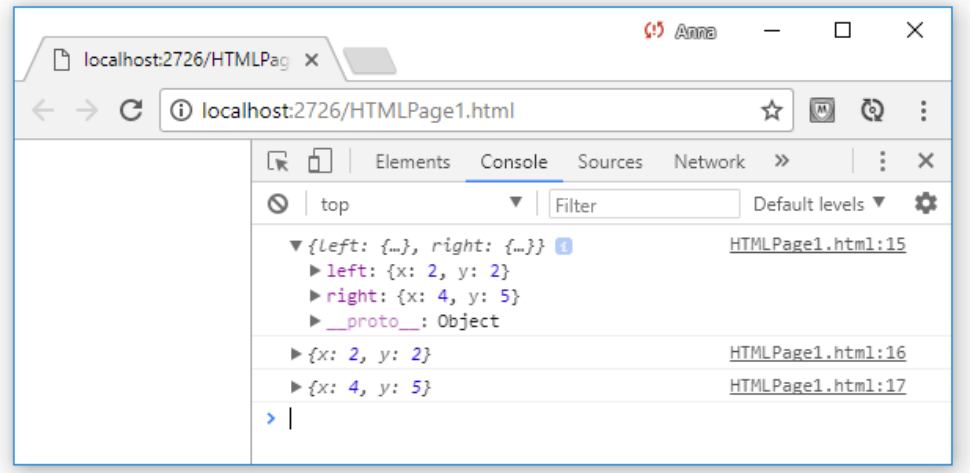
Object initializer הוא כמו array initializer, אך הסוגריים המרובעים מוחלפים בסוגריים מסולסלים, וכל subexpression מורכב ממפתח המאפיין + נקודותיים + ערך המאפיין. לדוגמה:

```
<!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>
```



Object literals יכולים להיות מקוננים. 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>
```



שימו לב: ניתן להוסיף לאובייקטים מאפיינים בצורה דינמית.

תרגיל

נתונה הפונקציה הבאה:

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

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

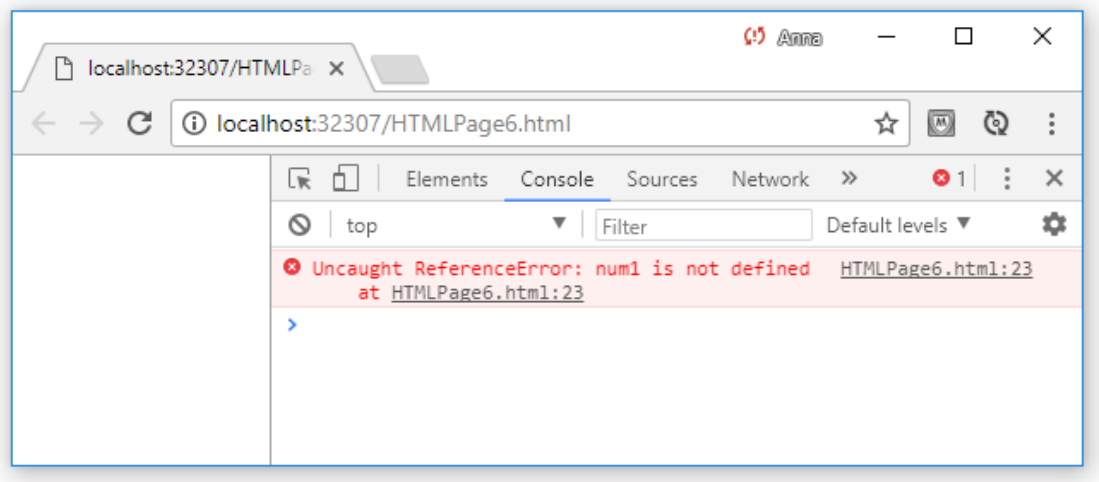
3. הגדרת משתנים – Variable Declaration

3.1. VAR

מאז היווסדה, ל- JavaScript הייתה דרך אחת להכריז על משתנים: var.
הצהרת משתנים באמצעות var, עובדת לפי עקרון ה- variable ופועלת כאילו המשתנים הוכרזו בראש ה execution context הנוכחי (פונקציה).
הדבר עלול 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 בדפדפן, נקבל את התוצאה הבאה:



עבור תוכניות גדולות - hoisting של משתנה יכול לגרום להתנהגות בלתי צפויה ולכן ECMAScript 2015 מציגה שתי דרכים חדשות להכרזה על משתנים:

- let
- const

3.2. let

הצהרת ה- let פועלת בדיוק כמו הצהרת ה- var, אך עם הבדל גדול: ההצהרות מוכרות רק בבלוק המקיף את המשתנה, וזמינות רק מהנקודה שבה ההצהרה ממוקמת.
לכן המשתנים המוצהרים על ידי let בתוך לולאה, או פשוט בתוך סוגריים מסולסלים, תקפים רק בתוך הבלוק הזה, ורק לאחר מכן תן הצהרה.
התנהגות זו היא הרבה יותר אינטואיטיבית. והשימוש ב let עדיף על השימוש ב- var ברוב המקרים.

כללים חשובים:

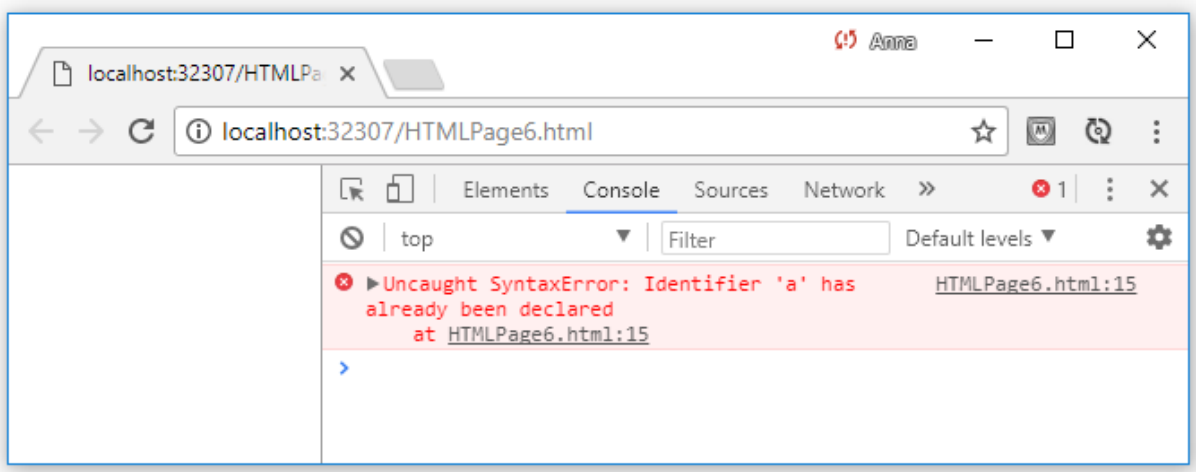
- **var** לא יכול להיות מוגדר פעמיים עם אותו שם בפונקציה אחת – (אפילו לא בלוק פנימי של הפונקציה) - למעשה, אנחנו יכולים להכריז פעמיים משתנה var, אבל זה לא יצור משתנה חדש, אלא עדיין יתייחס למשתנה בעל השם הזהה שהוגדר קודם לכן באותה הפונקציה.
- **let** - ניתן ליצור משתנה בעל שם זהה לבלוק החיצוני בתוך בלוק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

הנה כמה דוגמאות שיפשטו 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>
```

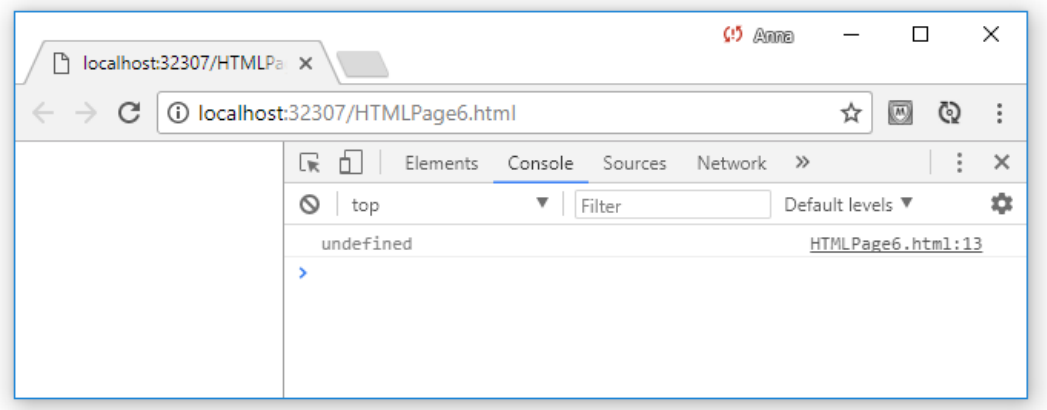
כאשר נריץ את הקוד בדפדפן, נקבל את התוצאה הבאה:



דוגמה שניה:

```
<!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>
```

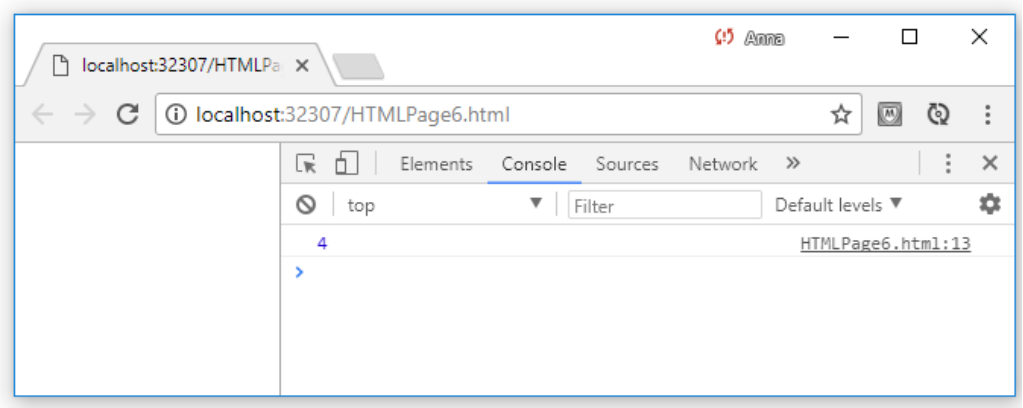
כאשר נריץ את הקוד בדפדפן, נקבל את התוצאה הבאה:



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>
```

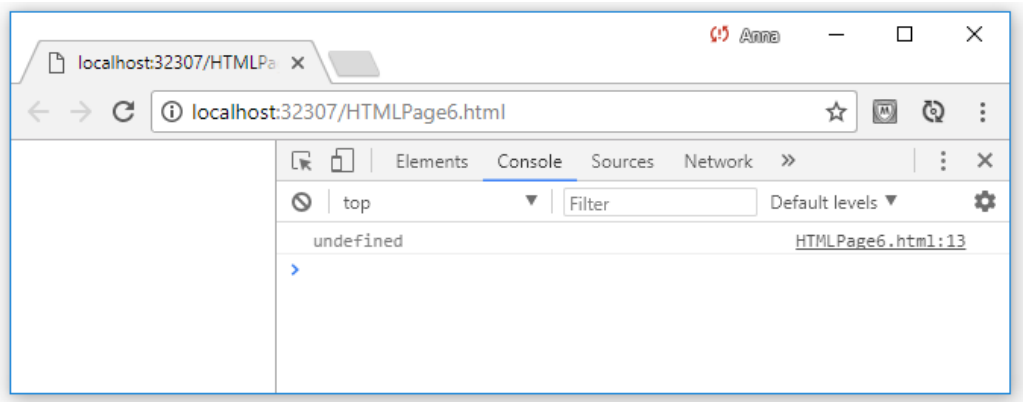
כאשר נריץ את הקוד בדפדפן, נקבל את התוצאה הבאה:



דוגמה 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>
```

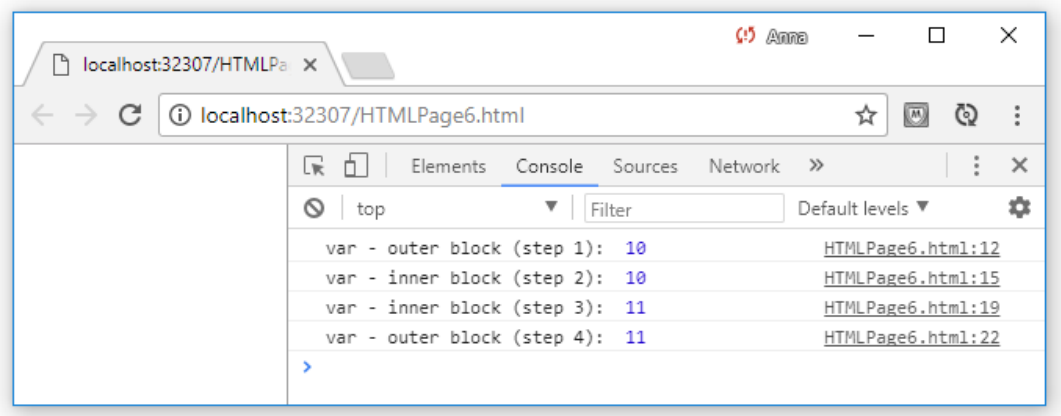
כאשר נריץ את הקוד בדפדפן, נקבל את התוצאה הבאה:



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>
```

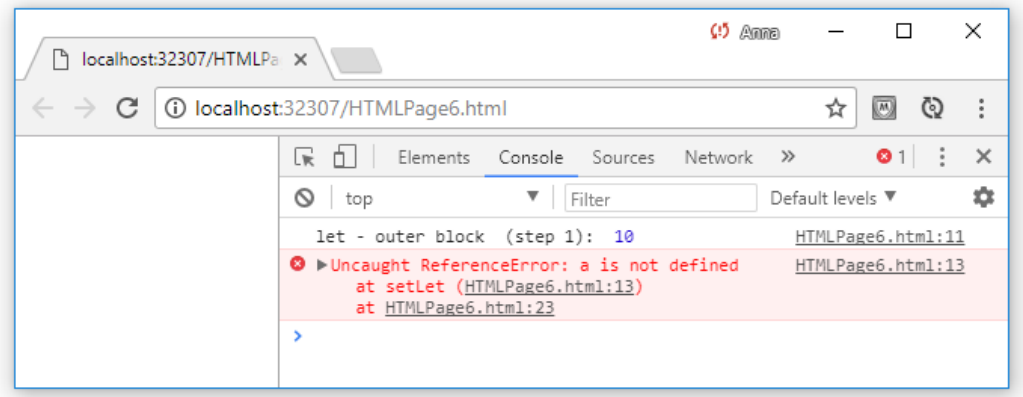
כאשר נריץ את הקוד בדפדפן, נקבל את התוצאה הבאה:



דוגמה 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>
```

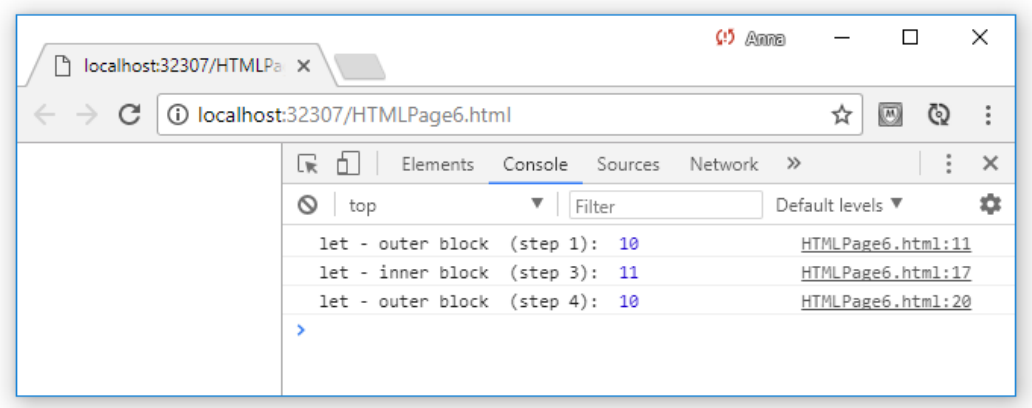
כאשר נריץ את הקוד בדפדפן, נקבל את התוצאה הבאה:



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>
```

כאשר נריץ את הקוד בדפדפן, נקבל את התוצאה הבאה:

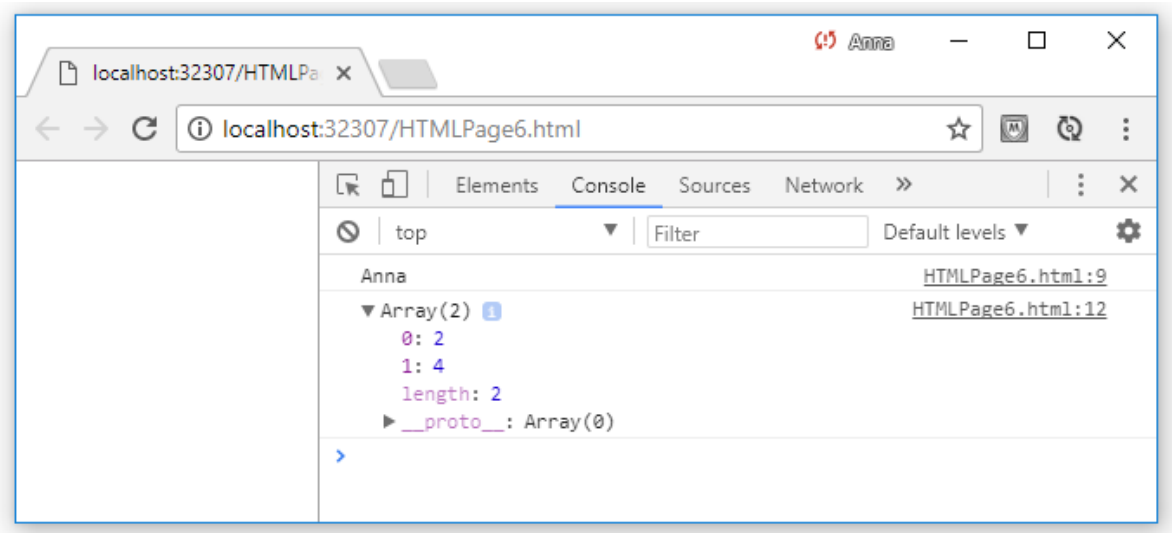


3.3. Const

הצהרת משתנה קבוע ב JavaScrip מתבצעת 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>
```

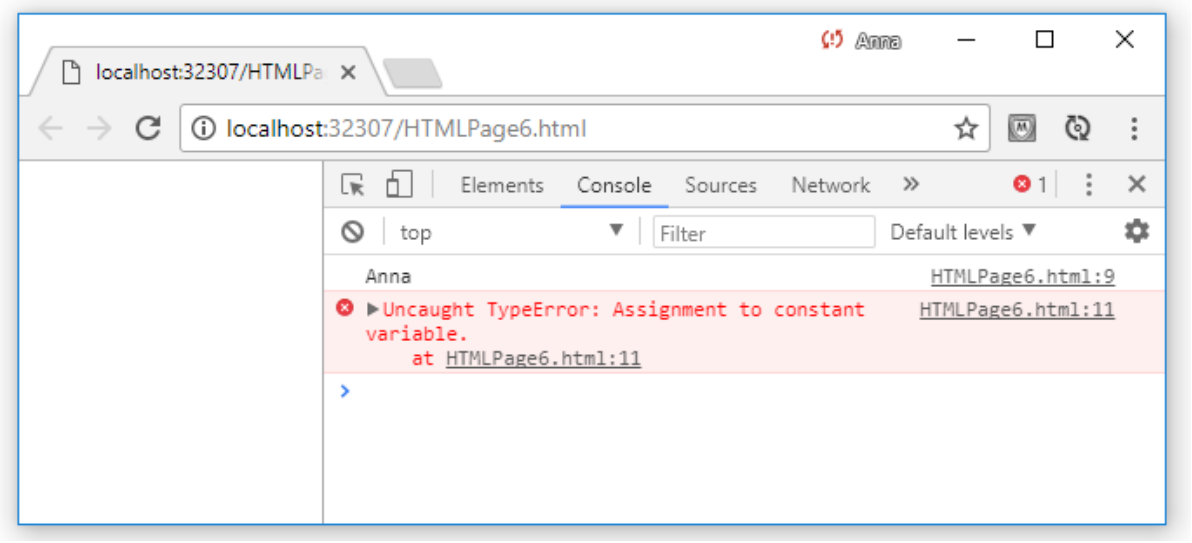
כאשר נריץ את הקוד בדפדפן, נקבל את התוצאה הבאה:



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

כאשר נריץ את הקוד בדפדפן, נקבל את התוצאה הבאה:

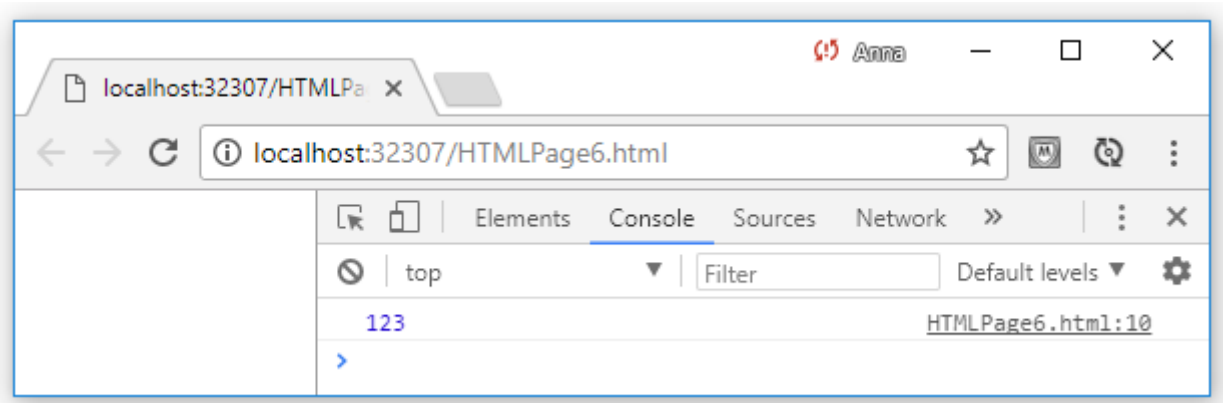


const אינו הופך את הערך ל immutable

Const פירושו שלמשתנה יש תמיד אותו ערך, אך אין פירוש הדבר שהערך עצמו הוא הופך להיות immutable (בלתי משתנה).
לדוגמה, obj הוא קבוע, אך הערך שהוא מצביע עליו הוא mutable (ניתן לשינוי) - ואנו יכולים להוסיף לו property (מאפיין):

```
<!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>
```

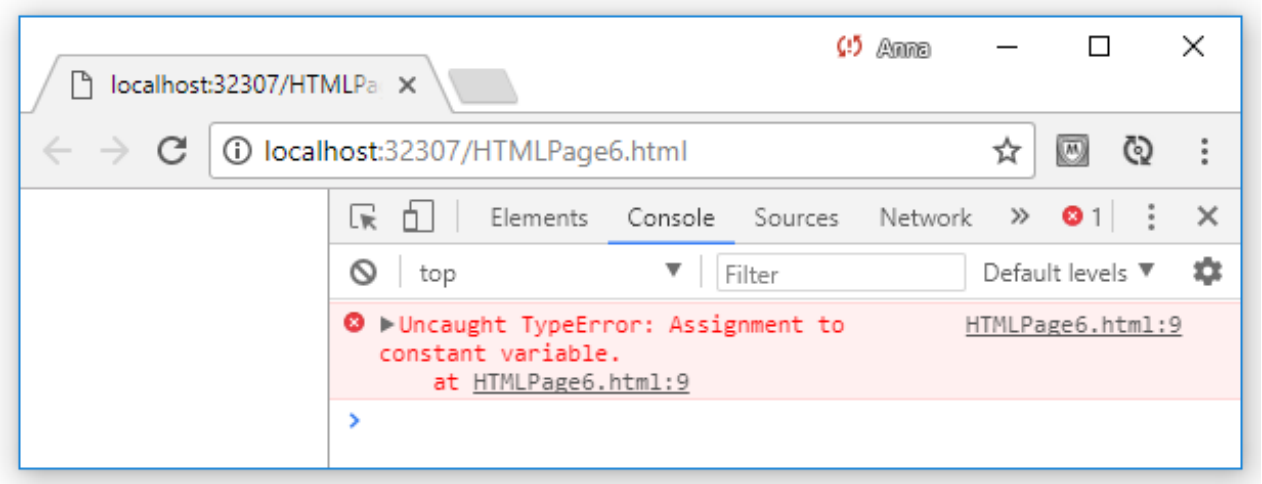
כאשר נריץ את הקוד בדפדפן, נקבל את התוצאה הבאה:



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 בדפדפן, נקבל את התוצאה הבאה:



Temporal dead zone .3.4

למשתנה שהוכרז על ידי let או Const יש אזור מת זמנית (TDZ):
כאשר הקוד מגיע לבלוק המקיף של אותו משתנה (scope), לא ניתן לגשת אליו (לקבל את תוכן השתנה או להגדיר ערך למשתנה) עד שתבוצע השורה של ההצהרה.

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

The life cycle of var-declared variables

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

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

The life cycle of let-declared variables

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

- כאשר הביצוע של הקוד מגיע ל-scope (הבלוק המקיף אותו) של משתנה let, נוצר שטח אחסון (כולל binding) עבורו. המשתנה נשאר uninitialized.
- גישה למשתנה במצב uninitialized גורמת ל - ReferenceError.

- כאשר הביצוע של הקוד בתוך ה- scope מגיע להצהרה, המשתנה מוגדר לערך שצוין על ידי האתחול (assignment) - אם קיים. אם לא קיים initializer אז הערך של המשתנה מוגדר ל undefined.

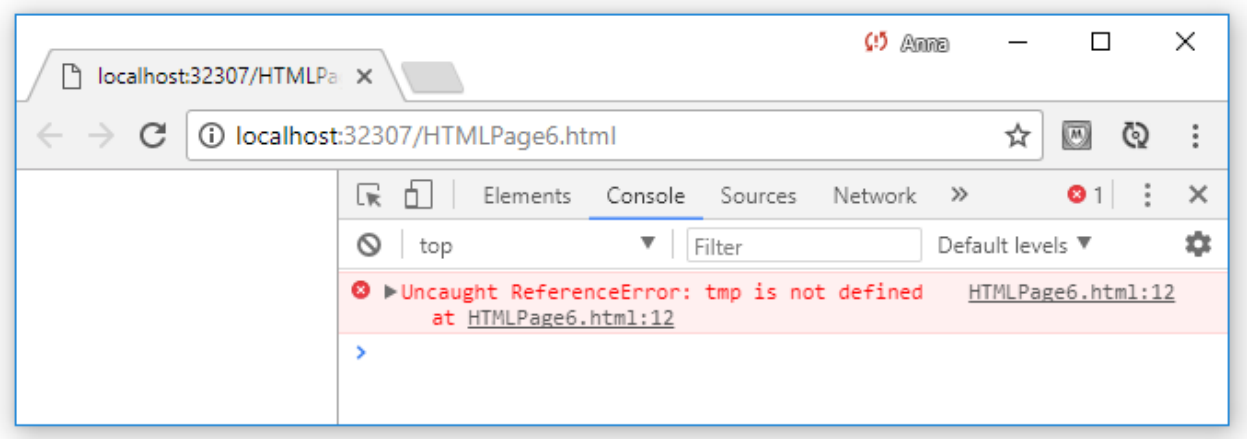
משתני const פועלים באופן דומה כדי למשתניlet , אבל הם חייבים להיות מאתחלים בשורת ההגדרה (כלומר, לקבל ערך מידי בשורת ההגדרה) ולא ניתן לשנות אותם בהמשך ה-scope.

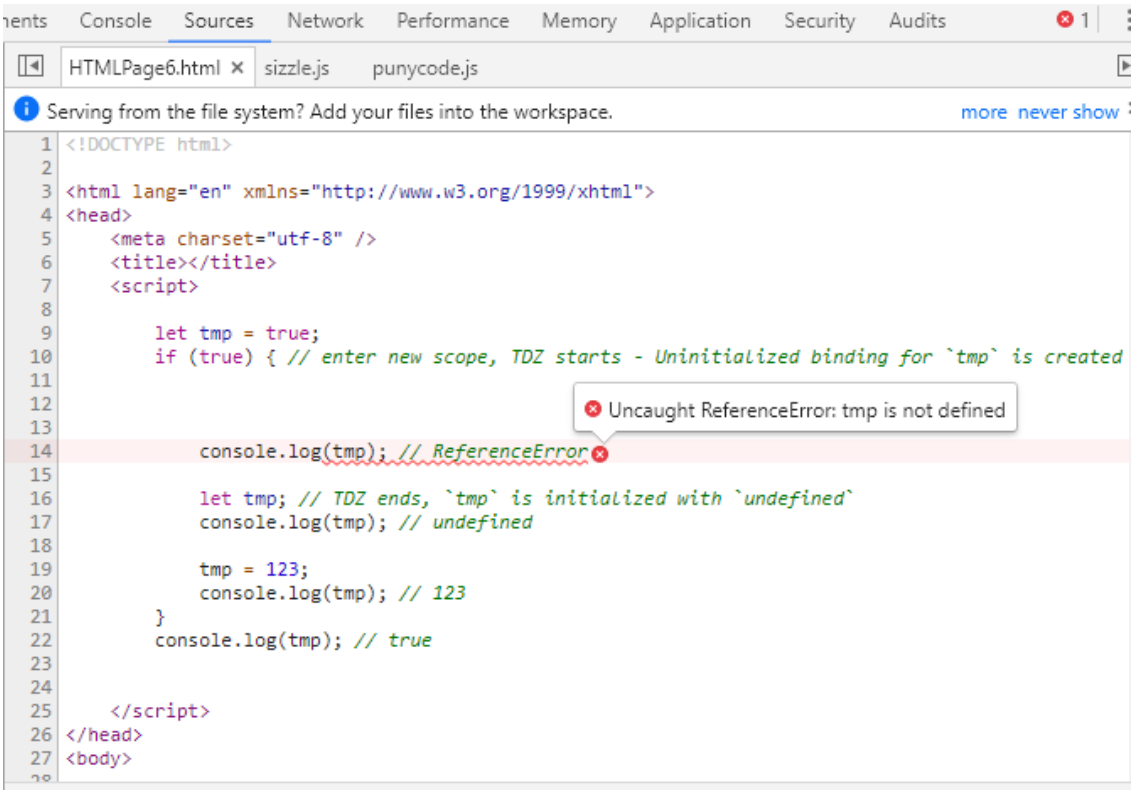
<u>דוגמאות:</u>

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

```
<!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>
```

כאשר נריץ את הקוד בדפדפן, נקבל את התוצאה הבאה:



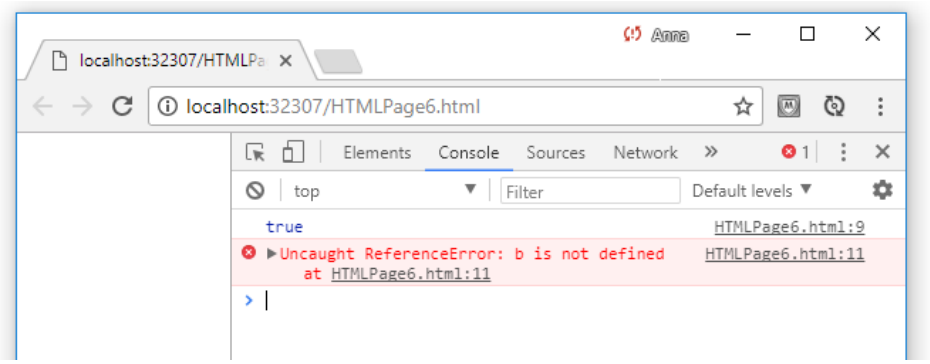


אם יש אתחול אז TDZ מסתיים לאחר ביצוע ה- initializer :

```
<!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>
```

כאשר נריץ את הקוד בדפדפן, נקבל את התוצאה הבאה:



הקוד הבא מדגים כי ה - dead zone הוא באמת זמני (על פי זמן) ולא מרחבי (על פי מיקום):

```
<!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>
```

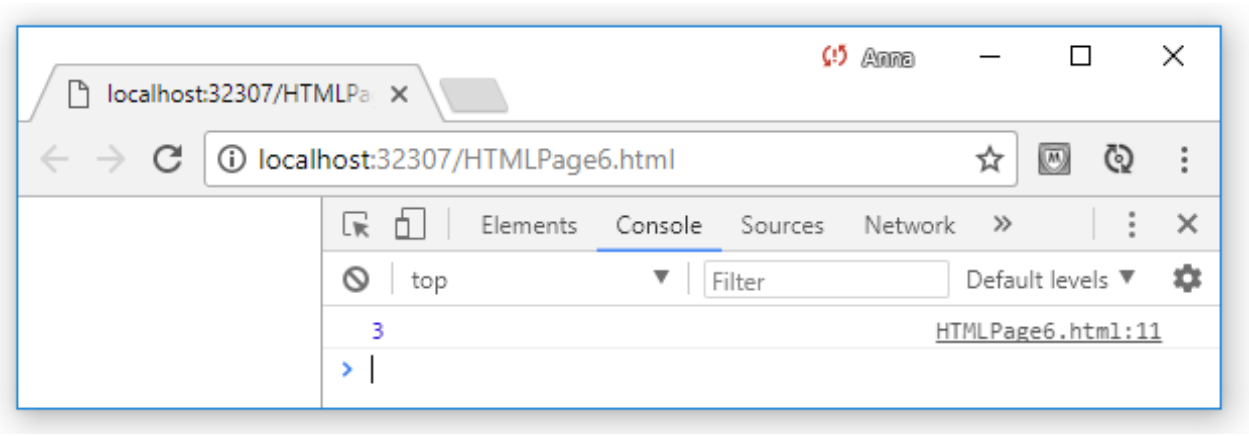
```
</html>
```

כאשר נריץ את הקוד בדפדפן, נקבל את התוצאה הבאה:


```
<!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>
```

כאשר נריץ את הקוד בדפדפן, נקבל את התוצאה הבאה:

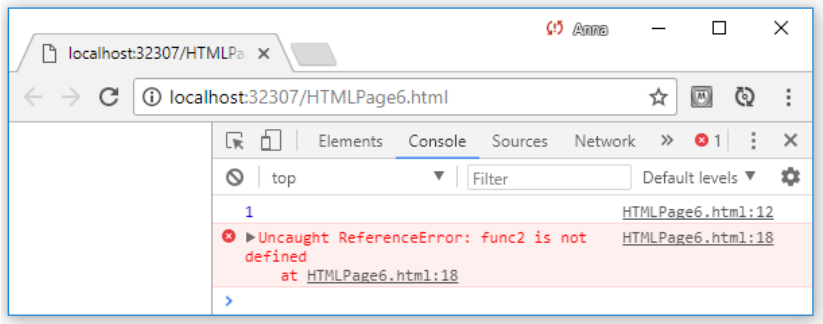


ערכי ברירת המחדל של פרמטרים ו temporal dead zone

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

```
<!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 בדפדפן, נקבל את התוצאה הבאה:



3.5. לולאות והגדרת משתנים

הלולאות הבאות מאפשרות לך להכריז על משתנים בראשם:

- for
- for-in
- for-of

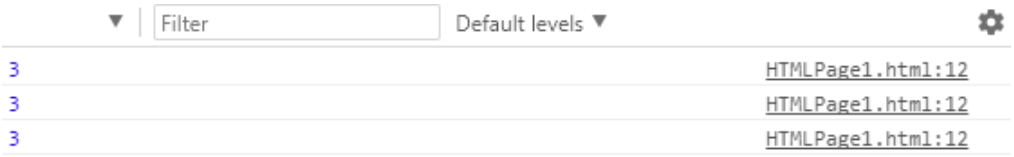
כדי להצהיר על המשתנים, אפשר להשתמש ב- let var או const.
כאשר לכל אחד מהם יש השפעה שונה.

for loop

var - הכרזה על משתנה בראש לולאה 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>
```

כאשר נריץ את הקוד בדפדפן, נקבל את התוצאה הבאה:

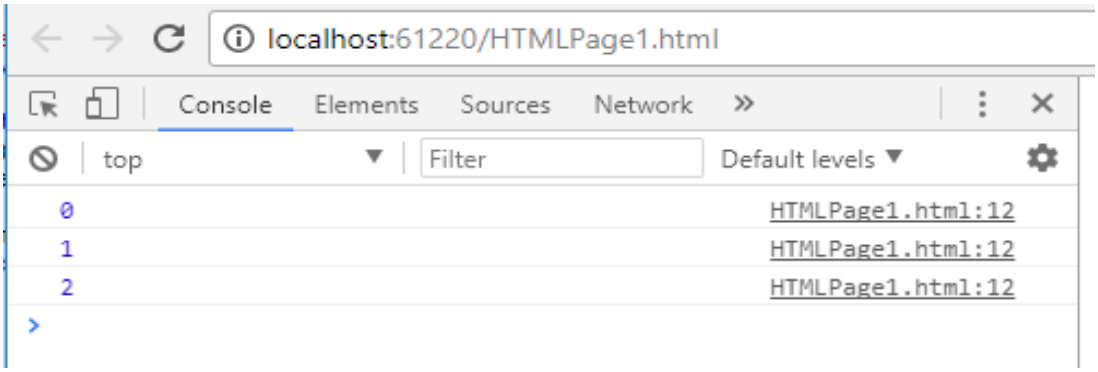


כל i בגופם של שלוש ה- arrow functions מתייחס לאותו binding, ולכן כולם יחזירו את אותו ערך.

אולם, אם הגדרת המשתנה תתבצע על ידי let, ייווצר binding חדש עבור כל איטרציה בלולאה:

```
<!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>
```

כאשר נריץ את הקוד בדפדפן, נקבל את התוצאה הבאה:

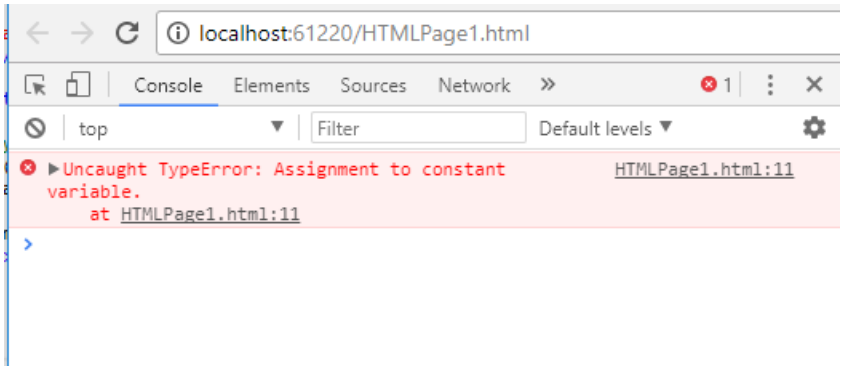


הפעם, כל אחד מתייחס ל binding של איטרציה מסוימת ומשמר את הערך שהיה קיים באותו זמן. לכן, כל arrow function מחזיר ערך שונה.

const עובד כמו var אבל אתה אי אפשר לשנות את הערך ההתחלתי 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>
```

כאשר נריץ את הקוד בדפדפן, נקבל את התוצאה הבאה:

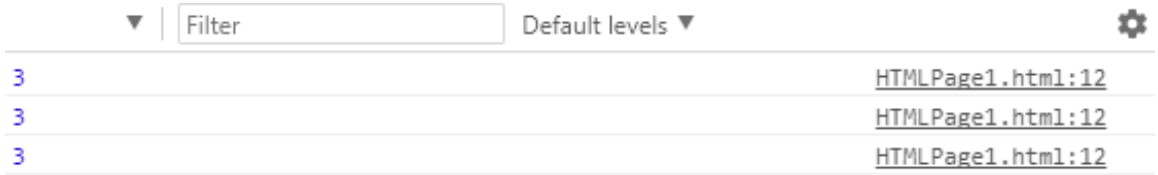


for-of loop and for-in loop

ב - for-of 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>
```

כאשר נריץ את הקוד בדפדפן, נקבל את התוצאה הבאה:

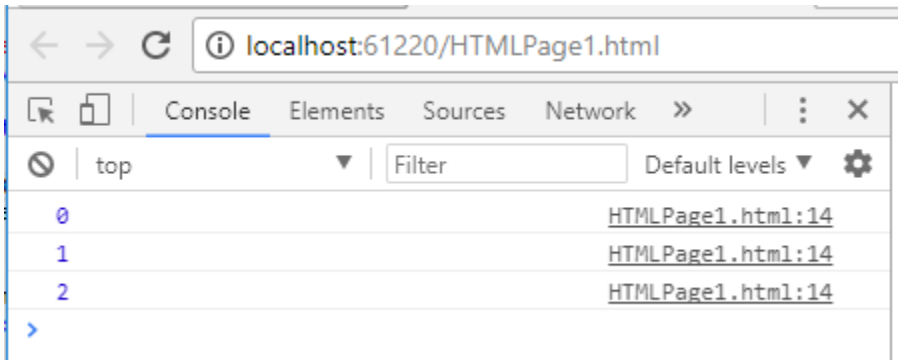


Const יוצר **immutable binding** אחד לכל איטרציה:

```
<!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>
```

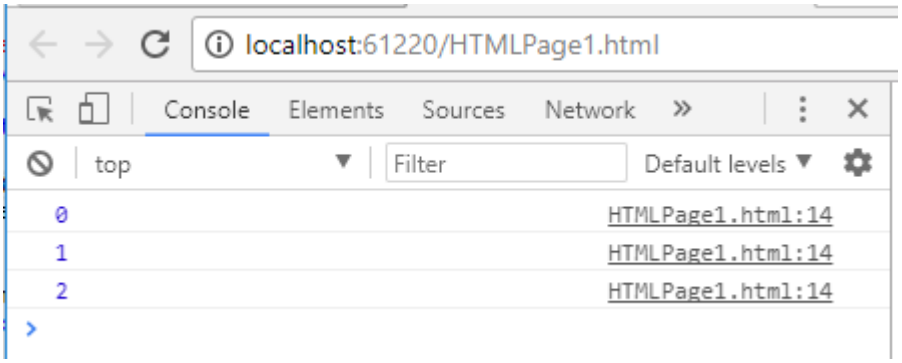
כאשר נריץ את הקוד בדפדפן, נקבל את התוצאה הבאה:



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

```
<!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:



לולאת for-in פועלת באופן דומה ללולאת for-of

3.6. סיכום אופני הגדרת משתנים

הטבלה הבאה מציגה סקירה של הדרכים בהן ניתן להגדיר משתנים ב- 6ES:

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

4. פונקציות

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

הגדרות פונקציית JavaScript יכולות להיות מקוננות בפונקציות אחרות, ויש להן גישה לכל המשתנים הנמצאים ב scopeשלהם, כאשר הם מוגדרים. משמעות הדבר היא כי פונקציות Java Script הם closures, וזוהי טכניקה חשובה, אותה נסקור בפירוט בהמשך הפרק.

4.1. דרכים להגדרת פונקציות

- functions as statement

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

- functions as expressions

שם הפונקציה הוא אופציונלי עבור פונקציות המוגדרות כביטויים. ביטוי הצהרת פונקציה למעשה מכריז על משתנה ומקצה לתוכו אובייקט פונקציה.
ביטוי להגדרת פונקציה:

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

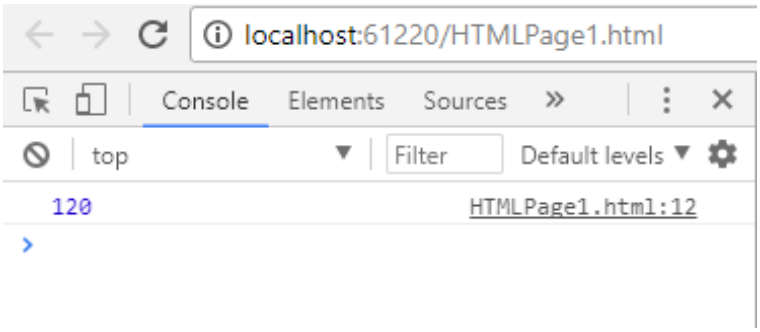
רוב הפונקציות המוגדרות כביטויים אינם זקוקים לשם פונקציה, מה שהופך את ההגדרה שלהם קומפקטית יותר.

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>
```

כאשר נריץ את הקוד בדפדפן, נקבל את התוצאה הבאה:

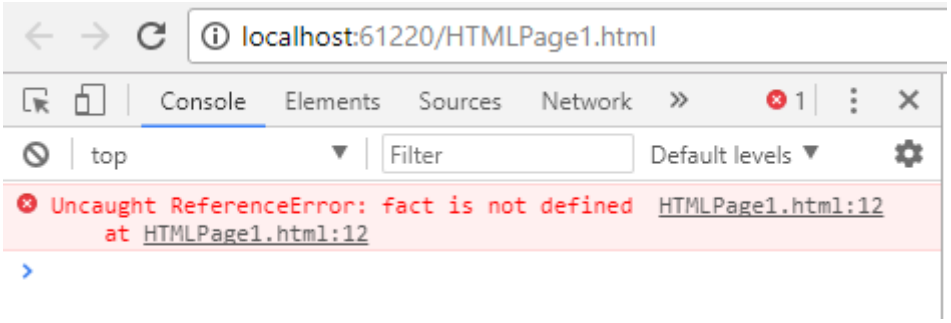


דוגמה זו צריכה להתייחס לעצמה. ולכן - אם ביטוי בהגדרת פונקציה כולל שם, היקף הפונקציה המקומית (local function scope) עבור פונקציה זו יכלול binding אל השם של אובייקט הפונקציה.
למעשה, שם הפונקציה הופך למשתנה מקומי בתוך הפונקציה.

אבל אם ננסה לקרוא לפונקציה fact() באמצעות השם של הפונקציה, מחוץ לקוד הבלוק של הפונקציה fact(), 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>
```

כאשר נריץ את הקוד בדפדפן, נקבל את התוצאה הבאה:



4.2. function hoisting

הצהרות הצהרת פונקציה "מועלות" לחלק העליון של ה- script המקיף או לחלק העליון של הפונקציה המקיפה אותם, כך שהפונקציות המוצהרות בדרך זו עשויות להיות מופעלות מקוד שמופיע לפני שהן מוגדרות.
כלל זה לא נכון עבור פונקציות המוגדרות כביטויים: כדי להפעיל פונקציה, חייבים להיות מסוגל להתייחס אליה, ואי אפשר להתייחס לפונקציה המוגדרת כביטוי עד ששורת ההגדרה של המשתנה בו היא מוקצה מתבצעת.

הערה:Variable declarations על ידי var מתבצעים בצורת hoisting בה המשתנים מוכרים כבר מראש הscope, אך ההשמות של הערכים למשתנים אלו אינן מועלות למעלה, ולכן לא ניתן להפעיל פונקציות שהוגדרו עם ביטויים לפני שהקוד ביצע את שורת ההגדרה.

4.3. Nested functions

ב- JavaScript, פונקציות יכולות להיות מקוננות בתוך פונקציות אחרות.

פונקציות מקוננות יכולות לגשת לפרמטרים ולמשתנים של הפונקציה (או הפונקציות) שהם מקוננים 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; }
```

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

        }

    </script>

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

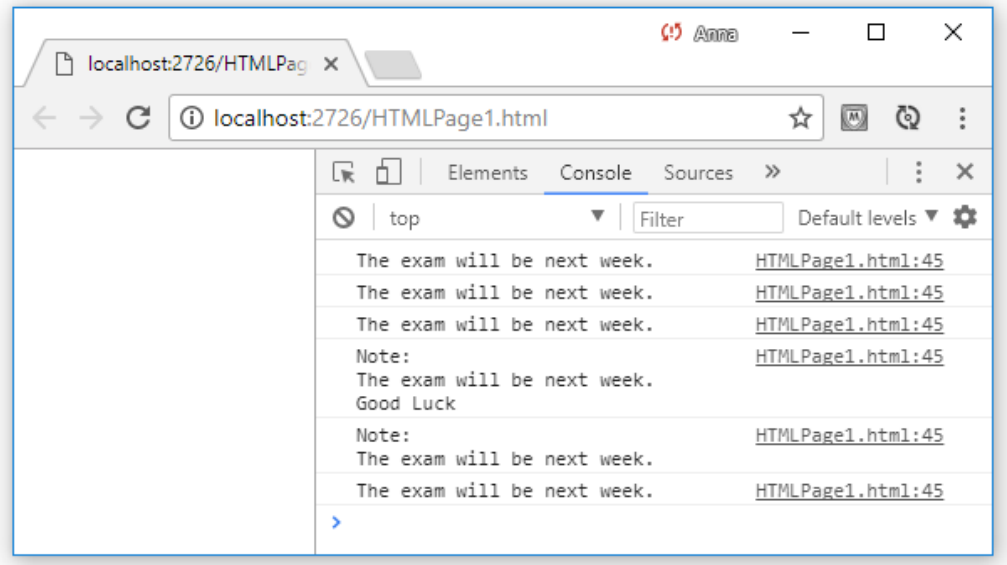
בקוד לעיל, הפונקציה הפנימית מרובע square() יכולה לגשת אל הפרמטרים a ו- b שהוגדרו על ידי הפונקציה החיצונית f() .

4.4. Function overloading

JavaScript , לא תומכת בצורה דיפולטיבית בהעמסת פונקציות, אולם ישנם דרכים לבצע "העמסה" מדומה. 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>

        // 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.");
```

```
        // 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>
```



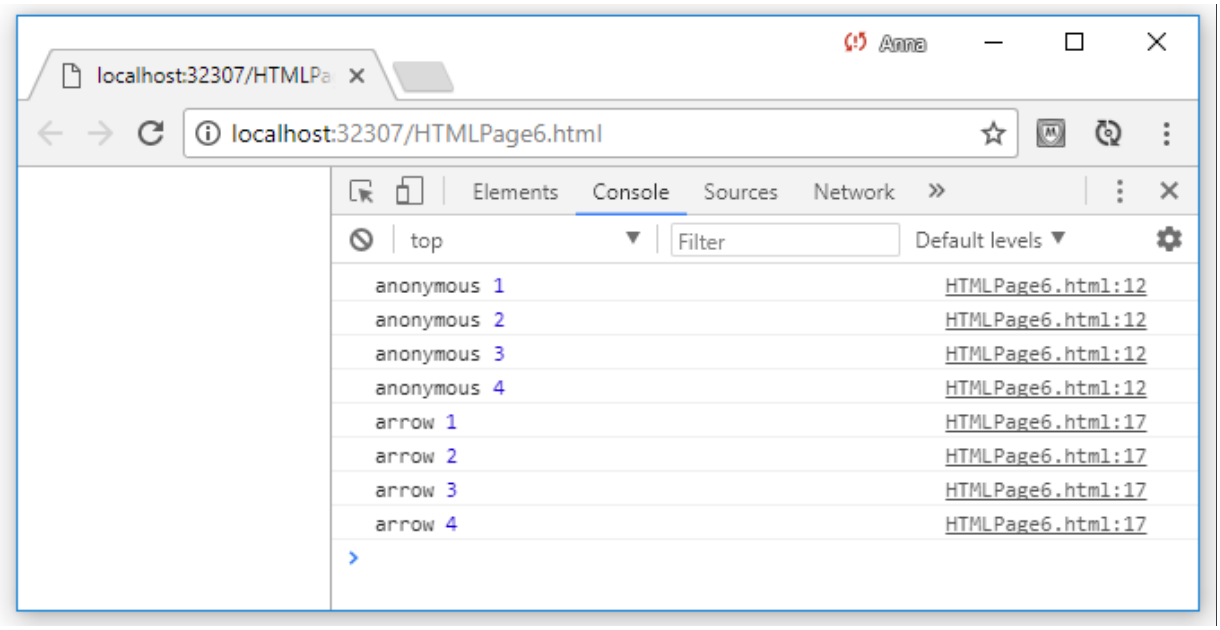
Arrow functions .4.5

JavaScript , על אף היותה multi-paradigm language, עושה שימוש בתכונות פונקציונליות רבות.
חלק מהתכונות הללו הן closures ופונקציות אנונימיות.

ב- 2015ES נוספה לתכונות 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>
```

כאשר נריץ את הקוד בדפדפן, נקבל את התוצאה הבאה:



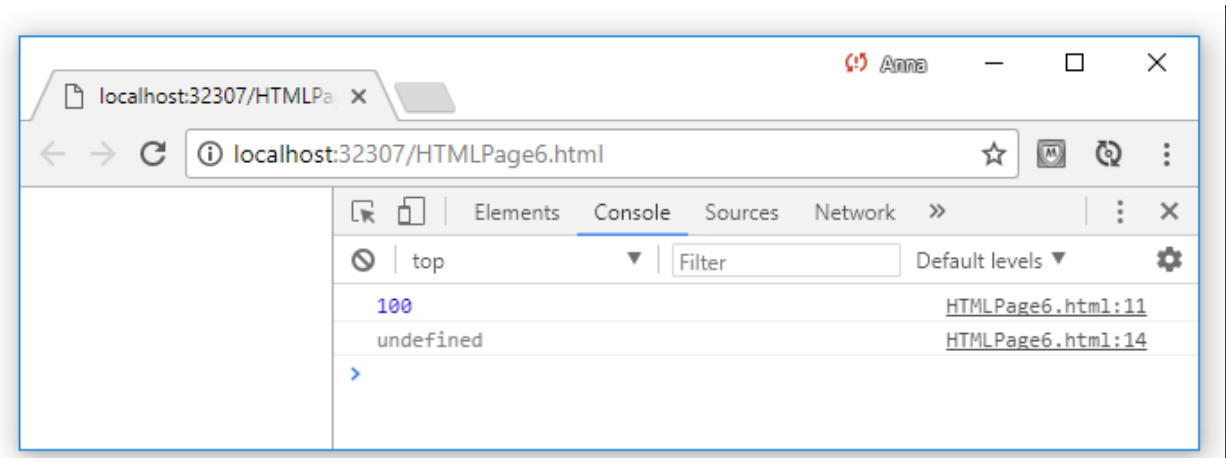
בהתחלה זה אולי נראה כמו שיפור קטן. עם זאת, arrow functions מתנהגות בצורה שונה כשמדובר ב-this.

arrow functions יורשות את הערך this מהפונקציה המקיפה אותם. בניגוד לפונקציות רגילות שלא יורשות את הערך this מהפונקציה המקיפה אותם, כפי שניתן לראות בדוגמא 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 בדפדפן, נקבל את התוצאה הבאה:

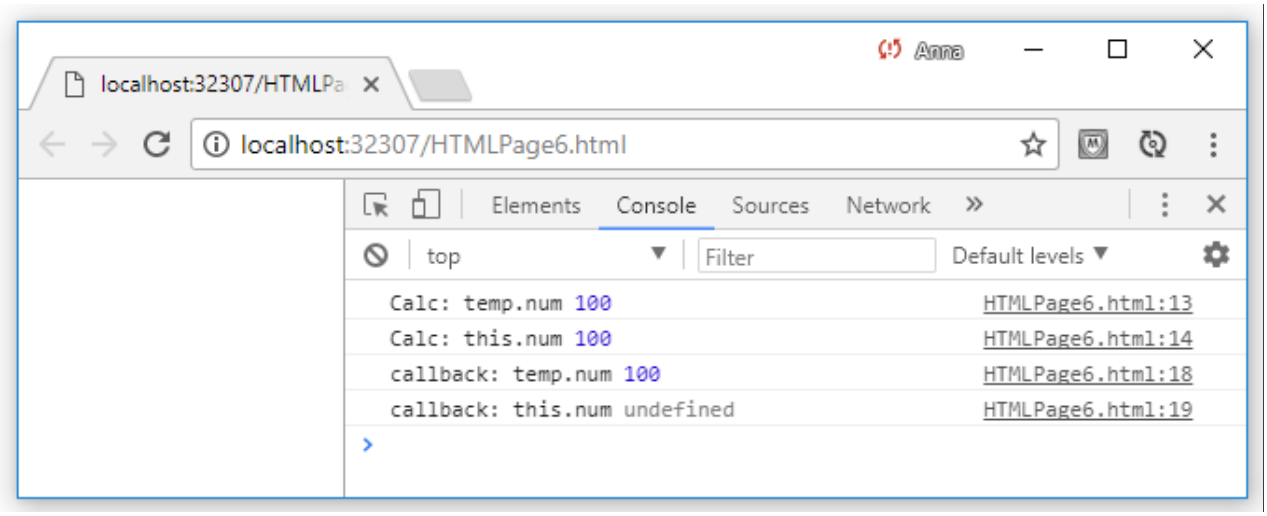


בגרסאות הקודמות, היה נהוג להתגבר על הבעיה הזו בדרך הבאה:

```
<!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>
```

כאשר נריץ את הקוד בדפדפן, נקבל את התוצאה הבאה:

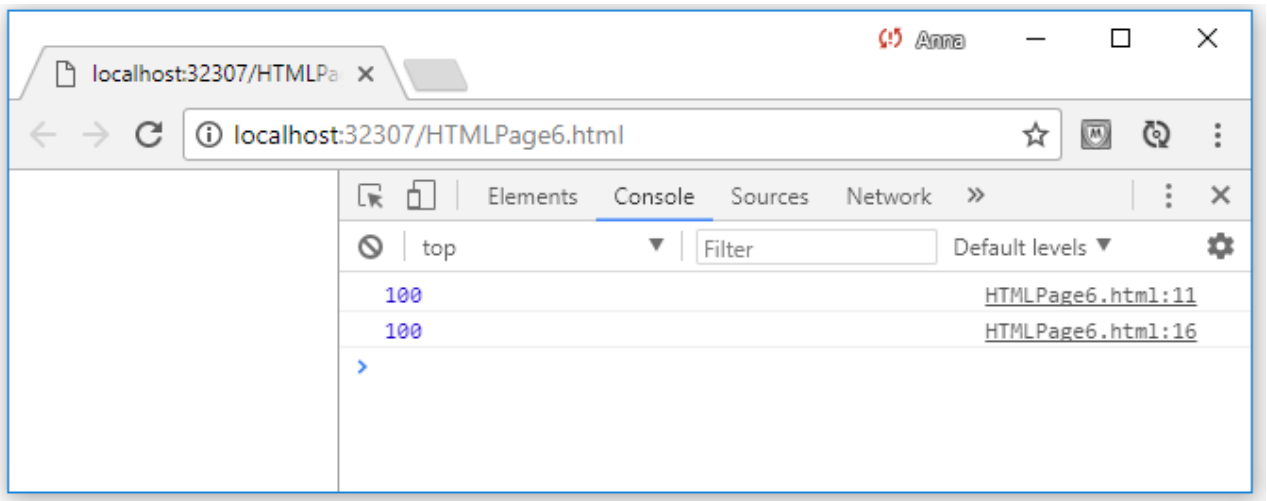


אולם, עם ECMAScript 2015 הדברים פשוטים יותר:

```
<!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 נריץ את הקוד בדפדפן, נקבל את התוצאה הבאה:



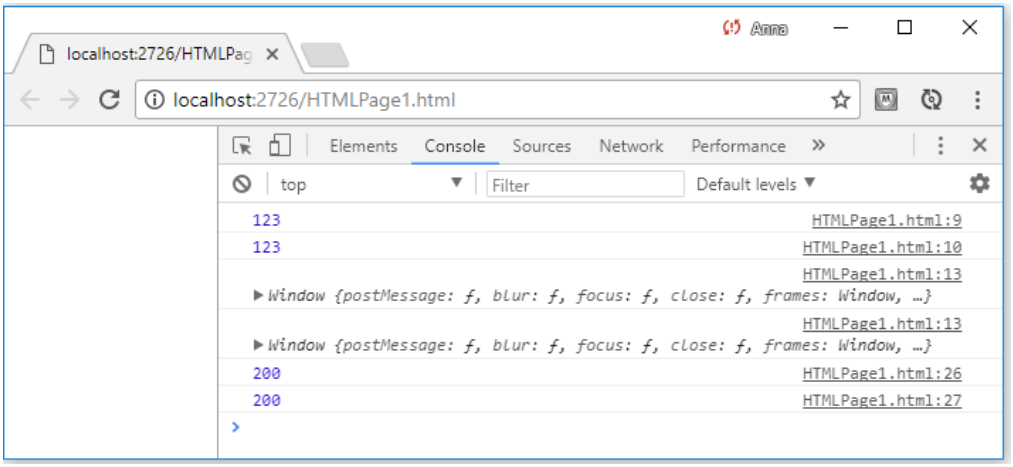
4.6. הגדרת משתנים גלובליים בפונקציה

כאשר נגדיר משתנים בפונקציה ללא המקדם var / let / const – אם הקוד לא נרשם במצב של strict mode המשתנה שנוצר מוגדר ברמה גלובלית – ונגיש 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 strict mode 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-invoke functions .4.7

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

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 שניה:

```
</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>
```

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

4.8. Closures

כמו ברוב שפות התכנות המודרניות - JavaScript משתמשת ב-lexical scoping.

משמעות הדבר היא כי פונקציות מבוצעות באמצעות variable scope שהיה בתוקף כאשר הם הוגדרו, ולא ה- variable scope הנמצא בתוקף כאשר הם מופעלים.

על מנת ליישם 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 ו scope (קבוצה של variable bindings) שבהם המשתנים של הפונקציה הם ה- variable scope הנמצא בתוקף כאשר הם מופעלים, נקרא closure.

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

רוב הפונקציות מופעלות תוך שימוש באותו scope chain שהיה בתוקף כאשר הפונקציה הוגדרה.

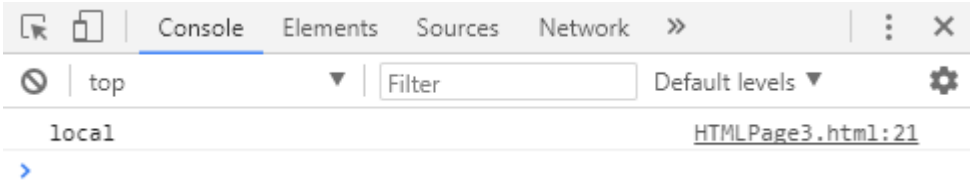
Closures הופכים מעניינים כאשר הם מופעלים תחת scope chain שונה מזה שהיה בתוקף כאשר הם הוגדרו. מצב זה קורה בדרך כלל כאשר אובייקט פונקציה מקוננת מוחזר מהפונקציה שבתוכה הוא הוגדר.

ישנן מספר טכניקות תכנותיות הכוללות את עקרון ה nested function closure , והשימוש בהן נעשה נפוץ יחסית בקוד JavaScript.

הצעד הראשון להבנת closures , הוא הבנת הכללים של lexical scoping עבור פונקציות מקוננות.
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>
```

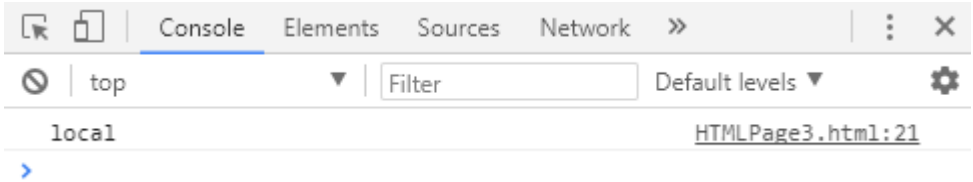


הפונקציה החיצונית outer() מכריזה על משתנה מקומי ולאחר מכן מגדירה ומפעילה פונקציה המחזירה את הערך של משתנה זה, וכפי שצפוי היא מחזירה "local".
עכשיו נשנה את הקוד, ובמקום להפעיל את הפונקציה המקוננת על ידי הפונקציה בתוכה היא מקוננת, נחזיר את הפונקציה המקוננת בתור ערך מוחזר בכל קריאה לפונקציה outer():

```
<!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>
```

כאשר אנו מפעילים את הפונקציה הפנימית מחוץ לאזור שבו היא הוגדרה, נקבל את התוצאה הבאה:



הסיבה ל output שקיבלנו היא הכלל הבסיסי של lexical scoping: פונקציות JavaScript מבוצעות באמצעות scope chain שהייתה בתוקף כאשר הן הוגדרו.

הפונקציה המקוננת inner() הוגדרה תחת scope chain של outer(), שבה ה- scope של המשתנה היה קשור לערך "local". וה- binding הזה עדיין בתוקף כאשר inner() מבוצע, לא משנה מהיכן הוא יבוצע.
זהו טבעם של closures: לשמר את ה- bindings של המשתנים הלוקליים והפרמטרים של הפונקציה החיצונית שבה הם מוגדרים.

ב- low-level programming languages כמו C, הארכיטקטורה של השימוש ב- CPU היא מבוססת מחסנית: אם משתנים מקומיים של פונקציה מוגדרים ב-stack CPU, הם יפסיקו להתקיים כאשר הפונקציה חזרה.

אבל ב- JavaScript אנו מגדירים scope chain בתור רשימה של אובייקטים (לא stack of bindings). ובכל פעם שמופעלת פונקציית JavaScript, נוצר אובייקט חדש שנועד להחזיק את המשתנים המקומיים עבור אותה קריאה, ואותו אובייקט נוסף לscope chain. כאשר הפונקציה חוזרת, ה- binding של האובייקט הזה מוסר מה-scope chain. אם אין פונקציות מקוננות, אזי אין עוד references binding לאובייקט והוא ישוחרר על ידי ה- garbage collector. אולם, אם הוגדרו פונקציות מקוננות, אזי לכל אחת מהפונקציות האלה יש references ל-scope chain, וה- scope chainמתייחס לאובייקט ה binding של הפונקציה החיצונית. אם אלה אובייקטים של פונקציות מקוננות והם נשארים רק בתוך הפונקציה החיצונית שלהם,

כאשר הפונקציה החיצונית תשוחרר על ידי ה- garbage collector הם גם ישוחררו, ולא ימשיכו להכיל binding לאובייקט הפונקציה החיצונית בה הם הוגדרו. אבל אם פונקציה חיצונית מגדירה פונקציה מקוננת ומחזירה אותה, המקום שביצע קריאה לפונקציה יכול לאחסן את הפונקציה הפנימית המוחזרת, ואז תהיה התייחסות חיצונית לפונקציה המקוננת. ולכן ה- garbage collector לא יוכל לשחרר את אובייקט הפונקציה החיצונית.

4.9. Arguments and parameters

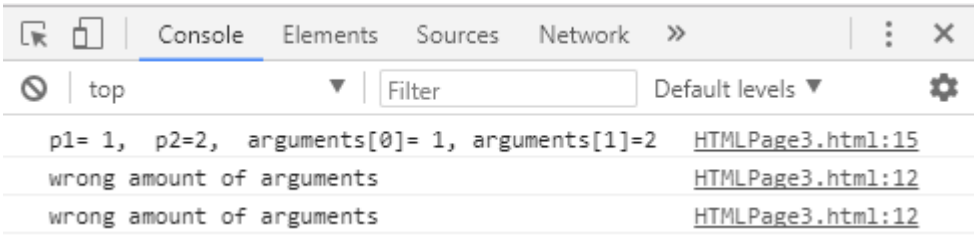
הגדרות פונקציית JavaScript אינן מציינות סוג נדרש עבור הפרמטרים של הפונקציות, ובעת קריאה לפונקציה, לא מתבצעת בדיקה כלשהי על ערכי הארגומנטים המועברים או על סוג הטיפוס שלהם. אבל כאשר פונקציה מופעלת עם ערכי ארגומנטים רבים יותר מאשר שמות פרמטרים, אין דרך להתייחס ישירות לערכים המיותרים.

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

האובייקט arguments הוא אובייקט דמוי מערך המאפשר לקבל את הארגומנטים שהועברו לפונקציה כדי לאחזר לפי מספר, ולא לפי 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>
```



הערה: במצב Strict-mode, arguments הם למעשה מילה שמורה. ופונקציות אינן יכולות להשתמש ב- arguments כשם פרמטר או כמשתנה מקומי, וכן לא ניתן לבצע השמה של ערכים ל arguments.

Invoking functions .4.10

ניתן להפעיל פונקציות **JavaScript** בארבע דרכים:

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

Function Invocation

פונקציות מופעלות כפונקציות או כשיטות עם ביטוי קריאה (סוגריים). בקריאה זו, כל ביטוי וארגומנט (אלה בין הסוגריים) מוערך, והערכים שהתקבלו הופכים לארגומנטים של הפונקציה. ערכים אלה מוקצים לפרמטרים הנקובים בהגדרת הפונקציה. ובגוף 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 שה interpreter הגיע אל סופה ללא שום פקודה המחזירה ערך, נערך המוחזר יהיה undefined.

Method Invocation

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

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

לאחר שהגדירנו את השיטה m עבור 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.

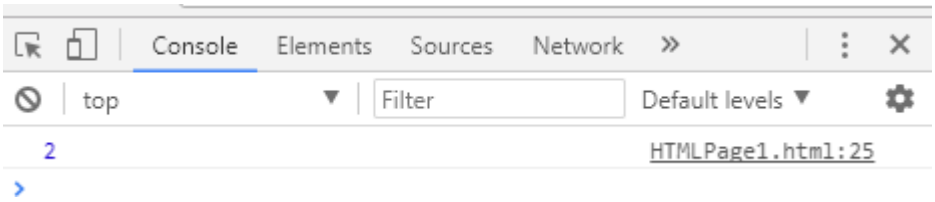
ביטויי גישה למאפייני אובייקט מורכבים משני חלקים:

- שם האובייקט
- שם המאפיין

כאשר המאפיין הוא method האובייקט איתו פנינו למאפיין הופך ל invocation context, וגוף הפונקציה יכול 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>
```



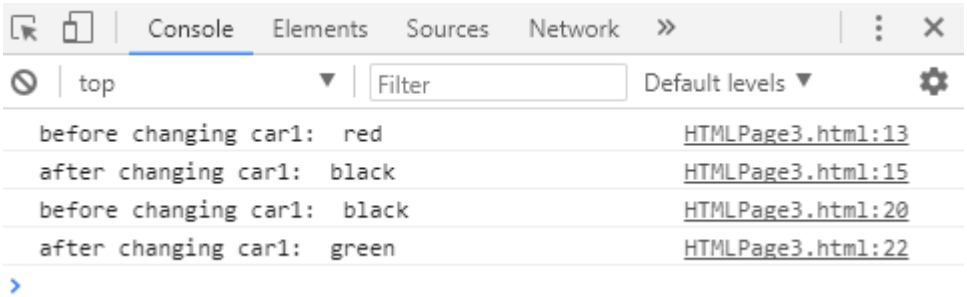
המונח Methods והמילה השמורה this , מהווים אבן חשובה לפרדיגמת התכנות מונחה העצמים. כל פונקציה המשמשת כשיטה מעבירה למעשה implicit argument - האובייקט שדרכו היא מופעלת

בדרך כלל, שיטה מבצעת איזושהי פעולה על האובייקט, והתחביר של method-invocation הוא דרך אינטואיטיבית להביע את העובדה שהפונקציה שנקראה פועלת על אובייקט.

בדוגמה הבאה נראה שתי שיטות לביצוע פעולה על אובייקט מסויים:

```
<!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);
        }
```

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


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

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



הפונקציות המופיעות בשתי שורות קוד אלו עשויות לבצע את אותה פעולה בדיוק על האובייקט 1car, אך התחביר של method-invocation בשורה הראשונה מבהיר על ידי ה syntax שלו את הרעיון שמדובר באובייקט 1car ועליו יתבצע ה operation.

הערה:this הוא keyword, ולא משתנה או property.

תחביר JavaScript אינו מאפשר להקצות ערך ל-this. אבל שלא כמו משתנים, ל- this keyword אין scope, ופונקציות מקוננות לא יורשות את הערך this של הפונקציה המכילה אותם.

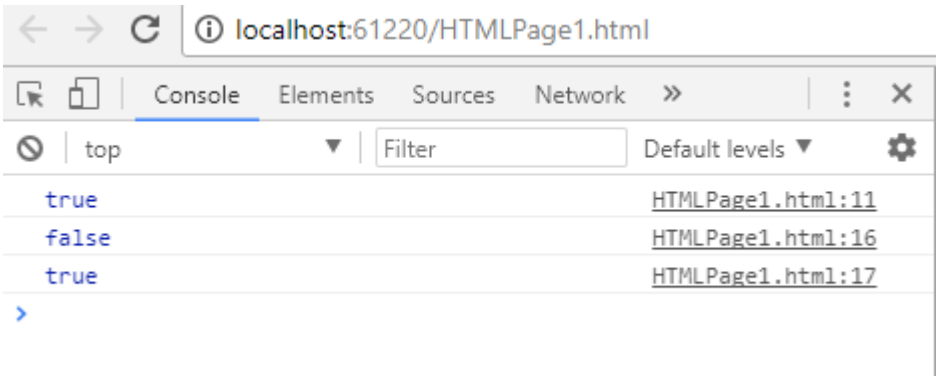
אם nested function מופעלת כשיטה, הערך this הוא האובייקט שהפעיל אותה. אולם אם פונקציה מקוננת מופעלת כפונקציה אז this יכיל את הערך global object (במצב non-strict mode) או undefined (במצב strict mode).

זוהי טעות נפוצה להניח כי פונקציה מקוננת המופעלת כפונקציה יכולה להשתמש ב- this כדי לקבל את ה- invocation context של הפונקציה החיצונית. אבל למעשה, אם רוצים לגשת לערך this של הפונקציה החיצונית, צריך לאחסן את הערך של this למשתנה אחר, כי הוא לא מוכר 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 () {
```

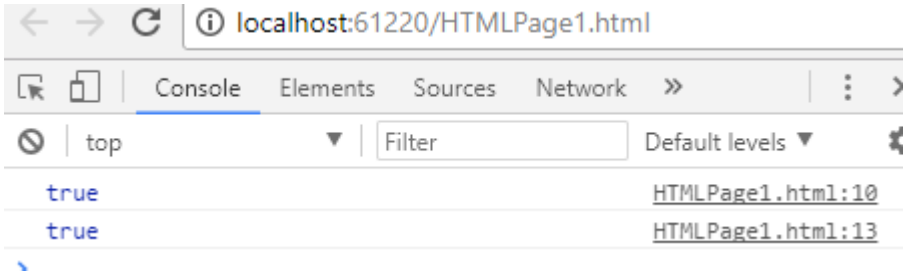
```
                var self = this;
                console.log(this === obj);

                f2();

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

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



או שאנחנו 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();
            }
        };
```

```
        obj.f1();     //Invoke the method

    </script>

</head>
<body>
</body>

</html>
```



Constructor Invocation

אם לפני הפניה לפונקציה התווספה המילה new, אז הקריאה היא constructor invocation
לכל פונקציה יש מאפיין prototype שמתייחס לאובייקט המכונה prototype object. לכל פונקציה יש prototype object שונה. וכאשר פונקציה משמשת כ constructor, האובייקט החדש שנוצר יורש מאפיינים מה prototype object של אותה פונקציה.
Constructor invocations נבדלים מ function ומ- method בטיפול ב arguments, ב -invocation context ובערך המוחזר.
אם הפניה ל constructor- כוללת רשימת ארגומנטים בסוגריים, ביטויים אלה מוערכים ומועברים לפונקציה באותה הדרך שבה הם יהיו עבור הפונקציות של פונקציות ושיטות.
אבל אם ל constructor אין פרמטרים, אז התחביר מאפשר להשמיט לחלוטין את הסוגריים של הקריאה 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 propertyשל הבנאי.
Constructor functions נועדו לאתחל אובייקטים והאובייקט החדש שנוצר משמש ל invocation 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 o אינו משמש כ invocation context.

Constructor functions אינן משתמשות בדרך כלל במילה return. מכיוון שתפקידם לאתחל את האובייקט החדש ולאחר מכן האובייקט הזה מוחזר בצורת return implicitly כאשר הבנאי הגיע לסוף הגוף שלו. אם בנאי השתמש במפורש בהצהרת return כדי להחזיר אובייקט, אז האובייקט הזה הופך להיות הערך של ביטוי 'הקריאה לבנאי,'אולם אם הבנאי מחזיר ערך פרימיטיבי, הערך הזה לא יוחזר בפועל, אלא תתבצע החזרה של האובייקט החדש המשמש בתור הערך של הפניה.

Indirect Invocation

פונקציות JavaScript הן אובייקטים וכמו כל האובייקטים של JavaScript יש להם שיטות.

שתי השיטות call() וapply() , מפעילות את הפונקציה בעקיפין.

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

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

הארגומנט הראשון של ()call()apply הוא אובייקט שבו יש לבצע את הפונקציה; ארגומנט זה הוא invocation context ובתוך הגוף של הפונקציה המילה השמורה thisתצביע עליו. כדי להפעיל את הפונקציה **f** () כשיטה של האובייקט **o** (ללא כל ארגומנטים), ניתן להשתמש באחת משתי הפקודות הבאות:

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

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

במצב strict mode בECMAScript 5 הארגומנט הראשון ()call ()apply מייצג את הערך של this, גם אם הוא ערך פרימיטיבי או nullאו undefined. ב- ECMAScript 3 וב non-strict mode , ערך של null או undefined מוחלף באובייקט הגלובלי וערך פרימיטיב-איטי מוחלף בwrapper object מתאים.

כל ארגומנטים שנשלחים ל- ()callלאחר הארגומנט הראשון הם הערכים המועברים לפונקציה שמופעלת. לדוגמה, כדי להעביר שני מספרים לפונקציה f ולהפעיל אותה כאילו היתה שיטה של אובייקט o, תוכל להשתמש בקוד כזה:

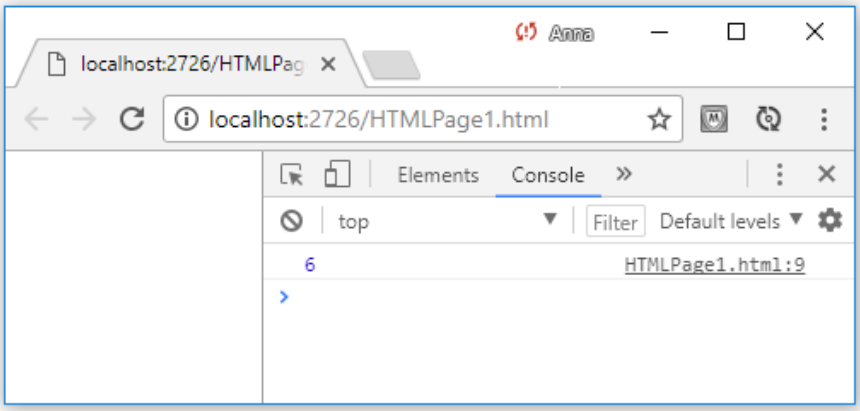
```
f.call(o, 1, 2);
```

השיטה ()apply דומה לשיטת , ()call אלא שהארגומנטים שיש להעביר לפונקציה מוגדרים כמערך:

```
f.apply(o, [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() כדי להעביר את מרכיבי המערך לפונקציה Math.max :

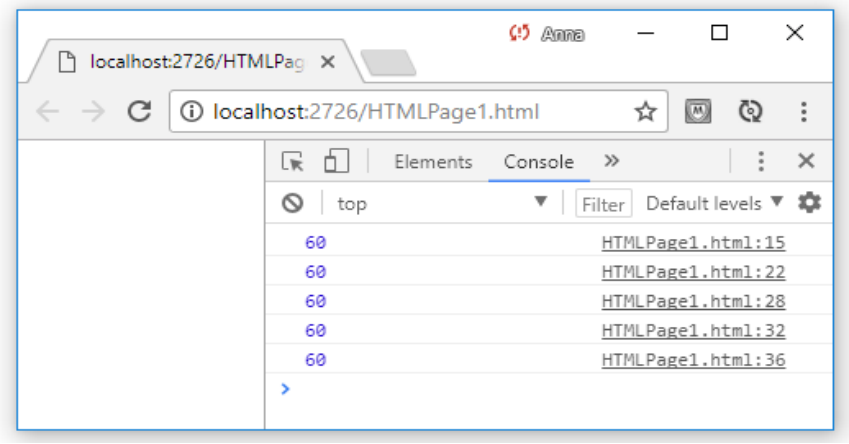
```
<!DOCTYPE html>

<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>
```



<u>לסיכום, ניצור פונקציה ונקרא לה בארבעת הדרכים שסקרנו לעיל:</u>

```
<!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;
        }
```

```
        //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

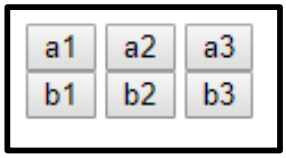
תרגילים בנושא closures

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 בדפדפן, נקבל את הדף הבא:



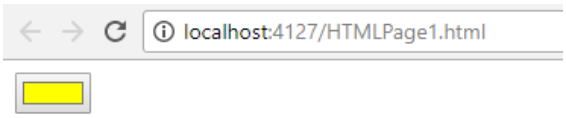
נסו לחשב (ללא הרצת הקוד – בהרצה יבשה) מה יהיה הפלט בלחיצה על כל כפתור

תרגיל 2

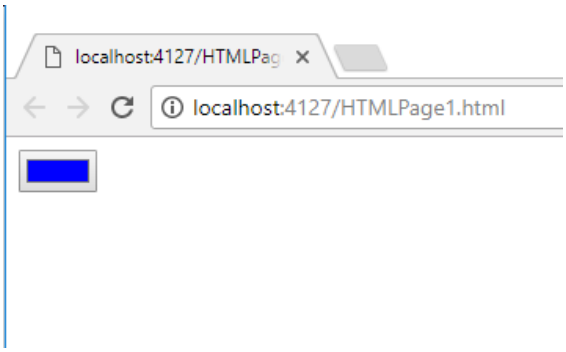
1. צרו בעמוד הHTML תיבת קלט מסוג COLOR
2. צרו קוד מתאים כך שבכל בחירת צבע של הלקוח תתבצע פונקציית setTimeout בDELAY של 5000 מילישניות, הפונקציה הזו תציג alert של הנתונים הבאים:
   - מספר הפעמים שהלקוח בחר צבע
   - המספר הסידורי של הבחירה הזו
   - הצבע האחרון שהלקוח בחר
   - הצבע שהלקוח בחר בבחירה הזו

לדוגמא:

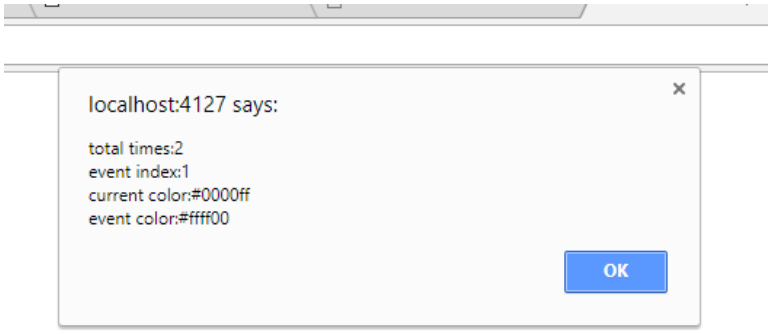
הלקוח ביצע בחירה ראשונה של צבע צהוב

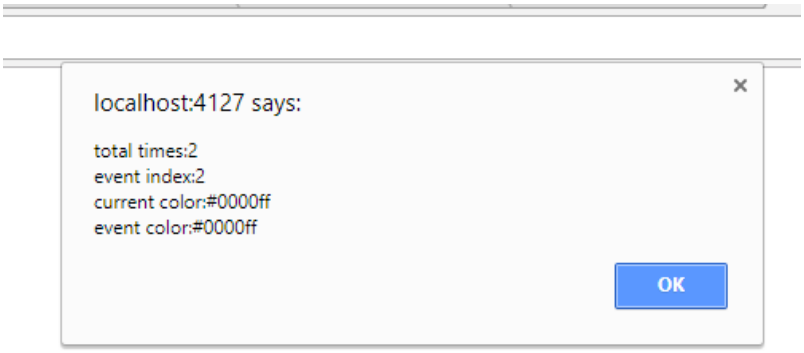


מיד לאחר מכן ביצע בחירה שניה של צבע כחול



לאחר חמש שניות יופיעו על המסך שתי ההודעות הבאות:





5. אובייקטים

5.1. מבנה האובייקט

אובייקט הוא unordered collection של properties, שלכל אחד מהם יש שם וערך.
שמות properties הם מחרוזות, כך שאנו יכולים לומר כי אובייקטים מפת מחרוזות לערכים.
אובייקטי JavaScript הם דינמיים, כך שניתן בדרך כלל להוסיף 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.
הmethods 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 לאף אובייקט לא יהיו שני מאפיינים בעלי שם זהה.

בנוסף לשם ולערך, לכל propertyיש ערכים משויכים שנקראים :property attributes

- התכונה writable מציינת אם ניתן לערוך את ערך המאפיין.
- התכונה enumerable מציינת אם שם המאפיין מוחזר על ידי לולאת .for/in loop
- התכונה configurable קובעת אם ניתן למחוק את המאפיין ואם תכונותיו ניתנות לשינוי.

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

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

Prototypes

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

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

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

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 ה host environment (כגון דפדפן אינטרנט) שבו מוטבע JavaScript. אובייקטי HTMLElement המייצגים את המבנה של דף אינטרנט ב- JavaScript בצד הלקוח הם host objects.
- host objects עשויים גם להיות native objects, לדוגמה כאשר ה host environment מגדירה שיטות שהן אובייקטי פונקציית JavaScript רגילים.

- **user-defined object**- אובייקט המוגדר על ידי המשתמש הוא אובייקט שנוצר על ידי ביצוע קוד JavaScript.

שני סוגים 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"יObject.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 רשימה בתוך סוגריים מסולסלים של מפתחות וערכים המופרדים בפסיקים ביניהם.
הנה כמה דוגמאות:

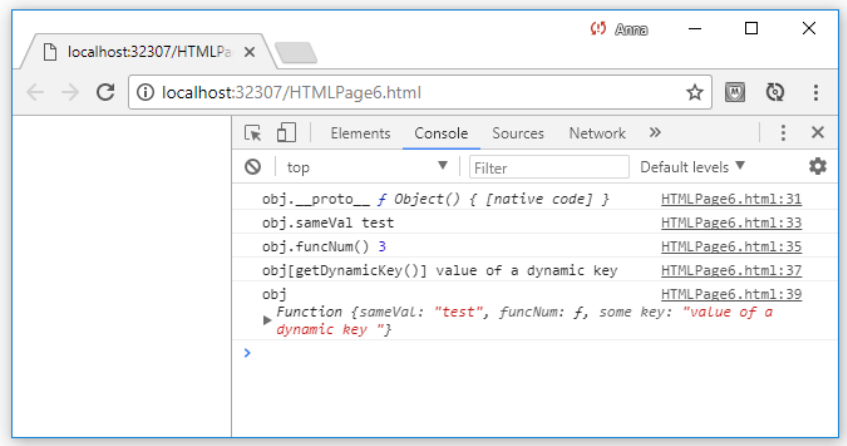
object literal הוא ביטוי שיוצר ומאתחל אובייקט חדש ונבדל בכל פעם שהוא מתבצע. הערך של כל נכס מוערך בכל פעם המילולית מוערכת. משמעות הדבר היא כי אובייקט יחיד של object literal יכול ליצור אובייקטים חדשים רבים אם הוא מופיע בתוך הגוף של לולאה בפונקציה הנקראת שוב ושוב, וכי ערכי המאפיינים של אובייקטים אלה עשויים להיות שונים זה מזה.

JavaScript Object-literal Improvements

2015ES הוסיפה 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,
```

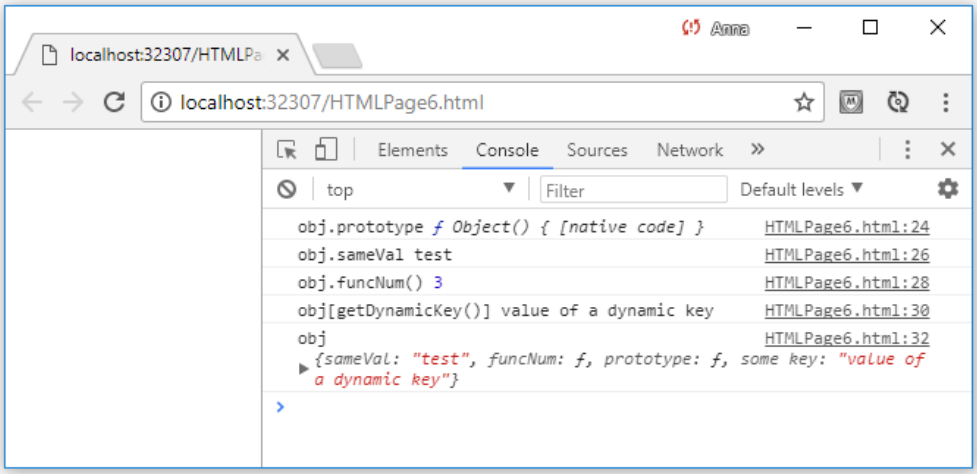
```
            // Methods can now be defined this way
            funcNum() {
                return 3;
            },

            // 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 הישנה הייתה דורשת את הדרך הארוכה 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 ומאתחל אובייקט חדש.

על המילה new להיות מלווה בפנייה לפונקציה. פונקציה המשמשת בדרך זו נקראת בנאי ומשמשת לאתחול אובייקט חדש.

Core JavaScript כוללת built-in constructors עבור native types. 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()

ECMAScript 5 מגדיר שיטה Object.create(), שיוצרת אובייקט חדש, באמצעות הארגומנט הראשון שלה כאב טיפוס של אובייקט זה.
Object.create גם לוקחת ארגומנט שני אופציונלי המתאר את המאפיינים של האובייקט החדש.

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

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

אפשר לשלוח את null כארגומנט, כדי ליצור אובייקט חדש שאין לו אב 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()), נעביר כארגומנט את הערך Object.prototype:

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

5.3. קריאת המאפיינים ושינוי האובייקט

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

אם נשתמש בנקודה, המילה הימנית חייבת להיות מזהה פשוט של אחד ממאפייני האובייקט. אם משתמשים בסוגריים מרובעים, הערך בתוך סוגריים 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

אובייקטי 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 ל-o 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 שיימצא ה x או עד שייערך חיפוש של אובייקט עם אב טיפוס ריק.

של אובייקט יוצר שרשרת או רשימה מקושרת שממנה prototype attributeכפי שניתן לראות, ה יורשים 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
```

עכשיו נניח שנרצה להשים ערך לתוך המאפיין x של אובייקט o. אם ל-o 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 מוסתר כעת על 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 לכלל הגדרת מאפיין באובייקט המקורי. והוא אם o יורש את המאפיין x, ואותו מאפיין הוא מאפיין accessor עם setter method, אז הsetter method נקרא במקום ליצור מאפיין חדש x ב-o.

Deleting Properties

האופרטור delete מסיר מאפיין מאובייקט.

האופרנד של האופרטור delete צריך להיות ביטוי גישה למאפיין. המחיקה 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 מחזירה את הערך- true אם המחיקה הצליחה או false אם למחיקה לא היתה השפעה (כגון מחיקת מאפיין שאינו קיים)
delete אינו מסיר מאפיינים בעלי configurable attribute בעל הערך false.

אופרטור in

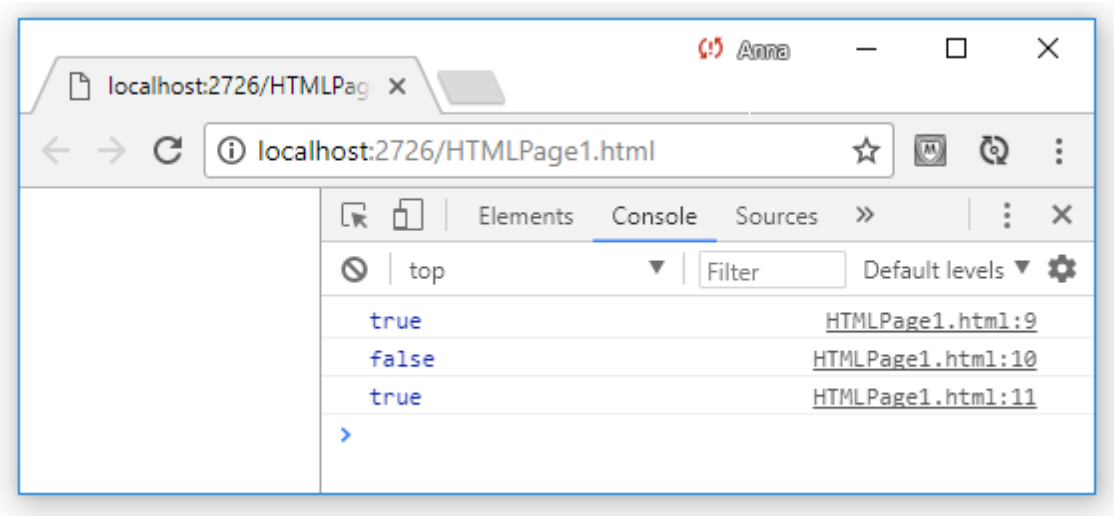
כדי לבדוק אם לאובייקט יש מאפיין עם שם נתון. ניתן להשתמש באופרטור in, או בשיטות:

- `hasOwnProperty ()`
- `propertyIsEnumerable ()`

אופרטור in מקבל משמאל את שם מאפיין (בתור מחרוזת) ומימין – את האובייקט שרוצים לבדוק האם המאפיין קיים בו ואז יוחזר true אם לאובייקט יש מאפיין פרטי או מאפיין בירושה בשם זה:

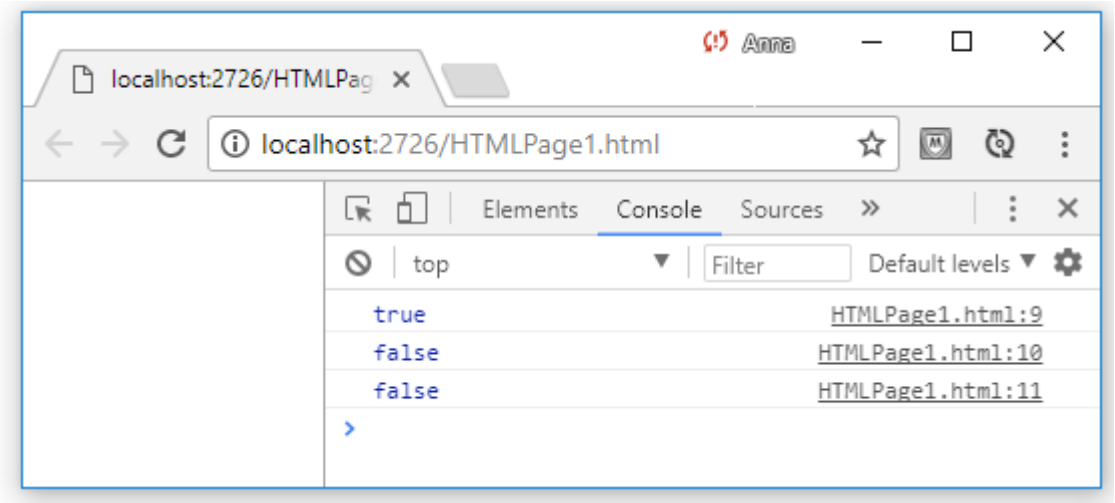
```
<!DOCTYPE html>

<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 משלו עם השם הנתון. היא מחזירהfalse 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>
```

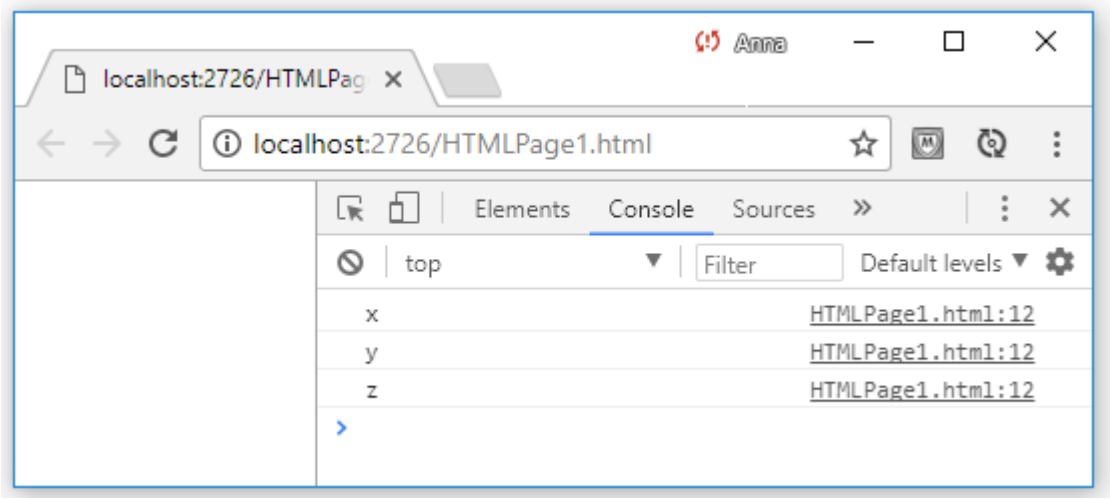


Enumerating Properties

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 שצוין, ומקצה את שם המאפיין למשתנה לולאה. שיטות מובנות שאובייקטים ירש לא יופיעו בצורה 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>
```



בנוסף ללולאת for/in, ECMAScript 5 מגדירה שתי פונקציות המחזירות את שמות מאפיינים.
הראשונה היא Object.keys(), המחזירה מערך של המאפיינים מסוג enumerable own properties של האובייקט.
השניה היא Object.getOwnPropertyNames(). שפועלת כמו Object.keys() אבל מחזירה את השמות של כל הownproperties של האובייקט שצוין, לא רק enumerable properties.

Property Getters and Setters

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

מאפיינים שהוגדרו על ידי getters ו setters ידועים לעתים כ accessor properties

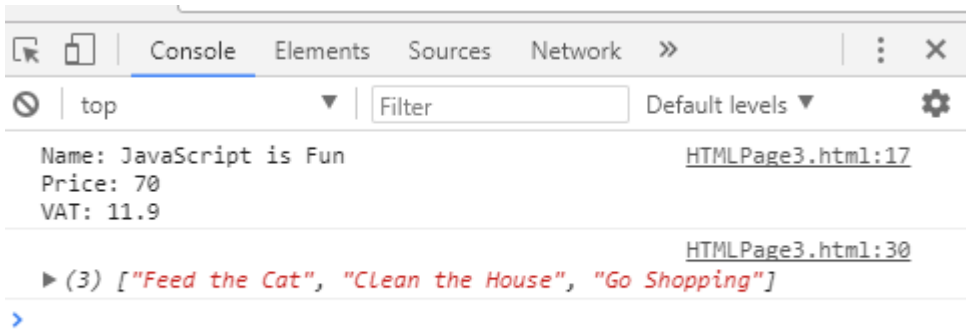
כאשר תוכנית ניגשת לקרוא את הערך של accessor property מפעילה את שיטת getter (לא מקבלת שום ארגומנטים). הערך המוחזר של שיטה זו הופך לערך של ביטוי הגישה למאפיין. כאשר תוכנית עורכת את הערך שלaccessor property היא מפעילה את שיטת setter, ומעבירה את הערך של הצד הימני של ההשמה.

ל Accessor properties אין writable attribute כמאפייני נתונים. אם מאפיין מכיל getter ו setter, זה המאפיין של קריאה / כתיבה.

הדרך הקלה ביותר להגדיר Accessor 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

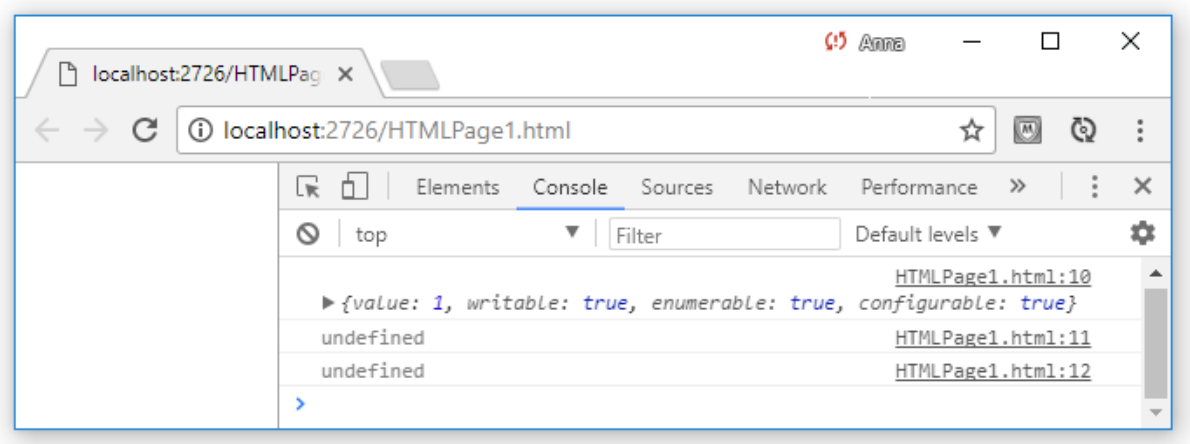
בנוסף לשם ולערך, למאפיינים יש attributes המציינים הרשאות עבור: written, enumerated, configured ב- ECMAScript 3 אין אפשרות להגדיר תכונות אלה: כל המאפיינים שנוצרו על-ידי תוכניות ECMAScript 3 ניתנים לכתיבה, , להגדרה, ולקבלה על ידי in, ואין אפשרות לשנות זאת. סעיף זה מסביר את ה- API של ECMAScript 5 להגדרת attributes של המאפיינים.

שיטות ה- ECMAScript 5 לקריאת ולקביעת המאפיינים של מאפיין משתמשות באובייקט הנקרא property descriptor כדי לייצג את קבוצת ארבע התכונות. אובייקט property descriptor כולל מאפיינים בעלי אותם שמות כמו התכונות של המאפיין, והם:

value, writable, enumerable, and configurable

ל- property descriptor יש get and set properties וכך אפשר לקבל ולשנות attributes של מאפיינים כדי לקבל את ה- property descriptor עבור מאפיין 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>
```



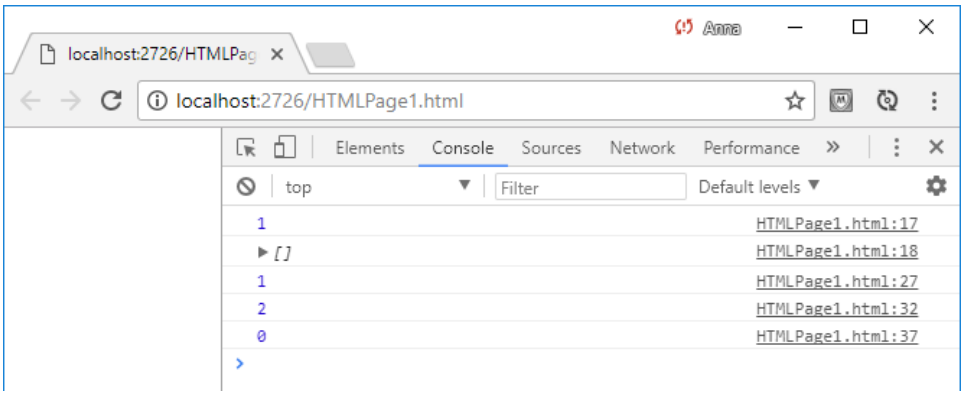
כדי להגדיר את ה attributes של מאפיין, או כדי ליצור מאפיין חדש עם attributes מוגדרים, יש להשתמש בפונקציה Object.defineProperty (), , המקבלת את הפרמטרים הבאים:

- פרמטר ראשון: שם האובייקט
- פרמטר שני – שם המאפיין שרוצים ליצור
- פרמטר שלישי – attributes עבור המאפיין שיוצרים

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
```

```
        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 ומכיל:
   - שם פרטי
   - שם משפחה
   - כתובת
   - מערך ציונים ריק
   - פונקציית setter המוסיפה ציון חדש למערך הציונים.
   - פונקציית getter המחזירה את ממוצע הציונים.

- פונקציית toString המחזירה ע"י return (ולא מציגה ע"י alert) את כל הפרטים של הסטודנט כמחרוזת אחת.

2. הוסיפו מספר ציונים ע"י פונקציית ה-setter.

3. קיראו לפונקציית ה-toString של האובייקט והציגו את המחרוזת המוחזרת ע"י alert.

6. אסינכרוני

תכנות סינכרוני אומר שאם שורת קוד 1 כתובה לפני שורת קוד 2, עד ששורת קוד 1 לא תסתיים לרוץ, שורת קוד 2 לא תפעל.

אפשר לתאר את זה כמו תור לקניית כרטיסים. עד שהאדם לפניך בתור לא יסיים, לא תוכל לקנות כרטיס בעצמך, ואותו הדבר לאדם שאחריך.

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, ששורה קוד 1 מתוזמנת לרוץ בהתאם לפעולה שתקרה בעתיד, ואז שורת קוד 2 תפעל לפניה.

אפשר לתאר את זה כמו הזמנה במסעדה : שולחן 1 מזמין מנה, ואחריו שולחן 2 מזמין 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. כלומר הגעת המנה מתוזמנת לאירוע שקורה (המנה מוכנה) ללא תלות בסדר הפעולות.

לאורך השנים פותחו בג׳אווה סקריפט מספר דרכים שבהם נוכל לתזמן חלקי קוד, להפעלה בעקבות פעולה שתוזמנה לעתיד.

הדרך הראשונה היא callback function

Callback function .6.1

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

פונקצית callback היא פונקציה שעוברת בתור ארגומנט לפונקציה אחרת, ואז execute בתוך אותה הפונקציה.

לדוגמה :

```html
<!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 insdie the callback function!");
        });
        console.log("I run first!");

    </script>

</body>

</html>
```

שימו לב לקוד : הפונקציה addEventListner היא בעצמה פונקציה (שימו לב לסוגרים () כשאנחנו קוראים לה) לתוכה אנחנו מעבירים 2 ארגומנטים, הראשון הוא מחרוזת "click" (שאומרת להאזין לאירוע של click), הארגומנט השני הוא פונקציה. שימו לב שהקוד של הפונקציה execute אחרי הקוד שיש בתוך הconsole.log , למרות שלפי סדר השורות הוא מופיע קודם. זה בגלל שהופנקציה מתוזמנת לרוץ רק אחרי שקורה אירוע של click. הפונקציה הזאת היא פונקצית callback, שעוברת בתור ארגומנט לפונקציה אחרת, וexecute בצורה אסינכרונית, ללא תלות ברצף השורות.

במשך שנים היה מקובל להשתמש בג׳אווה סקריפט אך ורק ב callback function בשביל לבצע פעולות אסינכרוניות. הבעיה הייתה שבגלל אופי השפה והשימוש הנרחב בפעולות אסינכרוניות, קוד שעושה שימוש נרחב בcall back , הופך להיות לא קריא ולא קל לתחזוקה. כפי שניתן 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))
          }
        })
      })
    }
```

וגרר אחריו מושג שנקרא call back hell

לכן נוצר הצורך לייצר כלי יותר נוח בשפה בשביל שימוש בפעולות אסינכרוניות. הכלי הזה נקרא Promise.

6.2. Promises

פרומיס הוא אוביקט שאמור להכיל ערך בזמן כלשהו בעתיד. כלומר עתיד להיות fullfill בצורה אסינכרונית. זהו אובייקט המתאר השלמת הפעולה האסינכרונית, או במקרה של שגיאה - דחיית הפעולה.

לפרומיס יש שלושה מצבים :

הראשון הוא Pending - שאומר שהערך של הפרומיס עדיין לא הוחלט. כלומר הפרומיס עדיין מחכה 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 fullfil

כאשר פרומיס הוא במצב של Pending , הוא אמור לשנות למצב של Fullfield או של Rejected, אבל אחרי שהוא שונה לאחד המצבים האלה, הוא לא יכול 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);
})
```

פרומיס אומנם היוו דרך יותר נוחה לשימוש בקוד אסינכרוני, אבל לאורך השנים גם שימוש נרחב בפרומיס הפך את הקוד לקשה לקריאה ולתחזוקה, 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 סקריפט מנגון אפילו יותר נוח לשימוש בקוד אסינכרוני בשם async await :

Async await .6.3

הרקע ליצירת המנגון של async await בשפה הוא שימוש בקוד קריא יותר. הקוד הוא אסינכרוני לכל דבר, מלבד העובדה שהוא כתוב כמו קוד סינכרוני, שורה אחר שורה. מה שמייצר דרך לכתיבת קוד אסינכרוני קריא ונוח יותר לתחזוקה.

נוכל 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 async נשים בתחילתה של הפונקציה. מה שהופך את הפונקציה ל - async function.

כפי שאפשר לראות בדוגמה, בשביל להמתין לערך החוזר מהפונקציה נשתמש ב then, בדיוק כמו השימוש שעשינו עם Promise.

נוכל גם להשתמש ב - async בשביל להחזיר Promise מהפונקציה:

```html
<!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 בawait יכול להיות רק בתוך פונקציה שהוגדרה בתור async, והיא משמשת על מנת לגרום לקוד "להמתין" לסיום של פעולה אסינכרונית כלשהי.

לדוגמה:

```html
<!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 "ממתין" ל Promise. הקוד נכתב בצורה המדמה קוד סינכרוני, אבל כמובן המשתנה יקבל את הערך שלו בצורה אסינכרונית לאחר מילוי ה Promise .

הכתיבה הנכונה וה best practice בשימוש ב async וה await , הוא "לעטוף" את await בתוך בלוק של try catch . פעולה אסינכרונית היא פעולה שמחכה להסתיים מתישהו בעתיד, ולכן קיים גם הסיכוי שפעולה זו תיכשל. בשביל "להגן" על הקוד שלנו ולהיות ערוכים לסוגים כאלה של כשלונות, וכן בשביל לנהל את השגיאות בצורה יותר נוחה. נרצה להשתמש בtry catch.

דוגמה :

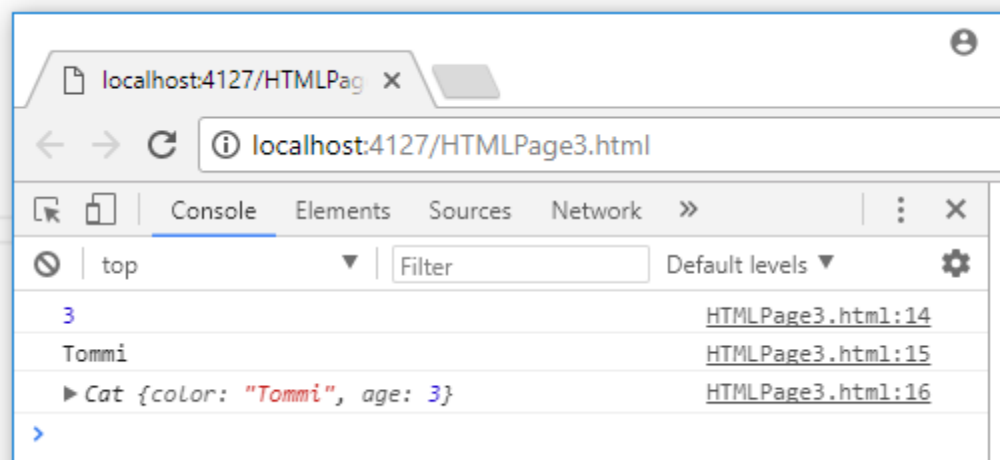
```
<!DOCTYPE html>
<html lang="en">

<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

7.1. Constructor function

אפשרי ליצור מעין "מחלקה" ממנה ניתן ליצור אובייקטים ע"י האופרטור new.
את הפונקציה ניצור כמו כל פונקציה רגילה – וניתן לה שם שמייצג את הטיפוס של האובייקטים שניצור ממנה בהמשך.
בתוך הפונקציה הזו – נרשום כל מאפיין או פונקציה שנרצה לאפשר לאובייקט שייווצר – על ידי המקדם this.
כאשר נקרא לפונקציה על ידי האופרטור new ייווצר אובייקט חדש – והמילה this תהיה הcontext של אותו אובייקט.
לדוגמא – ניצור constructor function של Cat שמכיל שני מאפיינים: color, age וניצור אובייקט ע"י הקריאה 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

נוכל להוסיף פונקציות שיתאפשרו להפעלה על ידי האובייקטים שייווצרו, בשני דרכים שונות:

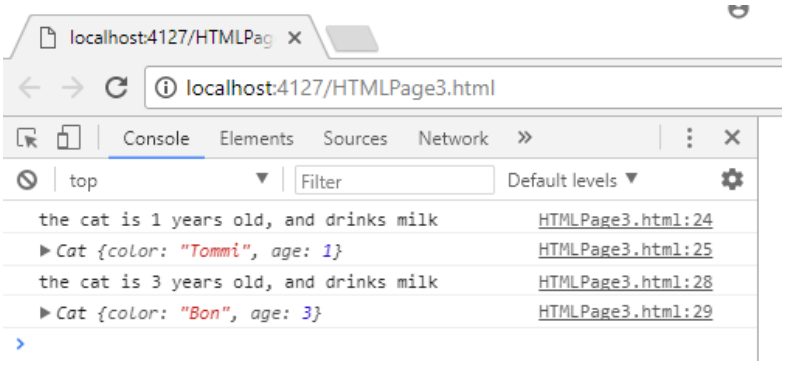
1. באותה הדרך בה הוספנו מאפיינים – כלומר: ניצור משתנה עם הקידומת this ולתוכו נאחסן פונקציה.
2. ע"י פניה מוץ לconstructor function אל הprototype של ה-constructor function – בדרך זו הפונקציה לא תווצר מחדש עבור כל אובייקט – ולכן בצורה זו ייחסך מקום בזיכרון.

שימו לב: בשתי הדרכים – השימוש במילה this בתוך הפונקציה, יתייחס לאובייקט הספציפי דרכו הופעלה הפונקציה.

להלן דוגמה המוסיפה 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 () { };
        */
    }
```

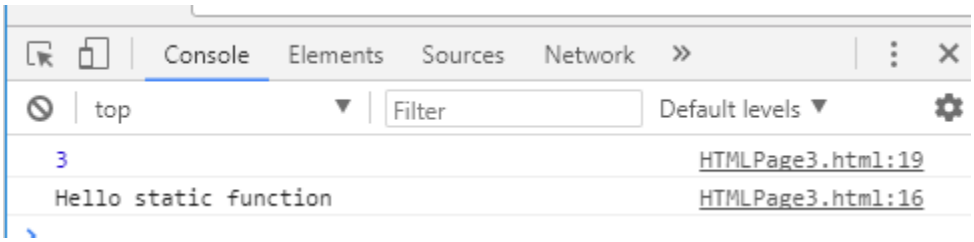
```
    //-----second way:
    // Won't duplicate function code to any future object, but will be placed at the
prototype once.
    Cat.prototype.drinkMilk = function () {
        return `the cat is ${this.age} years old, and drinks milk`;
    };

    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 protoype של הפונקציה – נוכל להוסיף לה משתנים ופונקציות סטטיות שיהיו נגישים בקוד ללא צורך ביצירת אובייקט של אותה הפונקציה על ידי new:

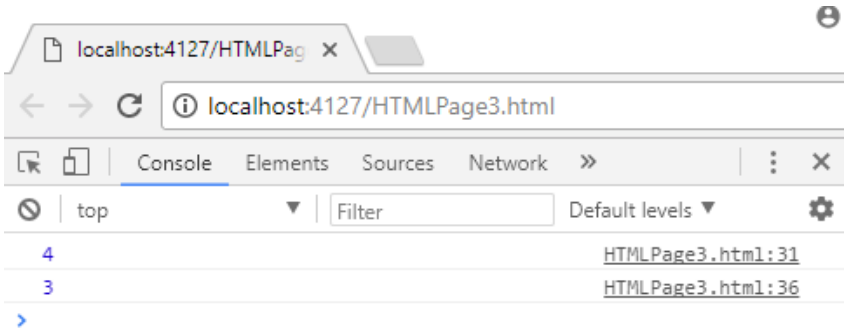
```
<!DOCTYPE html>

<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 private ל - Constructor Function

בכדי לממש את עקרון הencapsulation המורה כי על מחלקות לשמור על כללי כימוס, ולבצע בדיקות ואלידציה עבור משתנים באופן פנימי, נוכל להוסיף ל-constructor function משתנים פרטיים, באופן הבא:

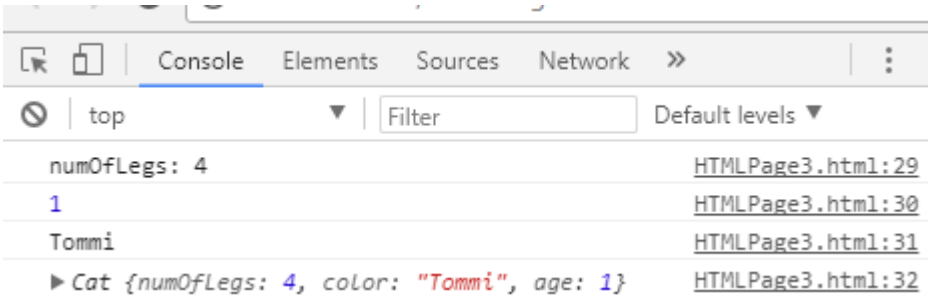
```
<!DOCTYPE html>

<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);
    c1.setAge(-4);
    console.log(c2.getAge());


</script>
</head>
<body>

</body>
</html>
```



Inheritance

אם נרצה ליצור function constructor של Animal ולהגדיר שCat יורש מ,Animal 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>
```

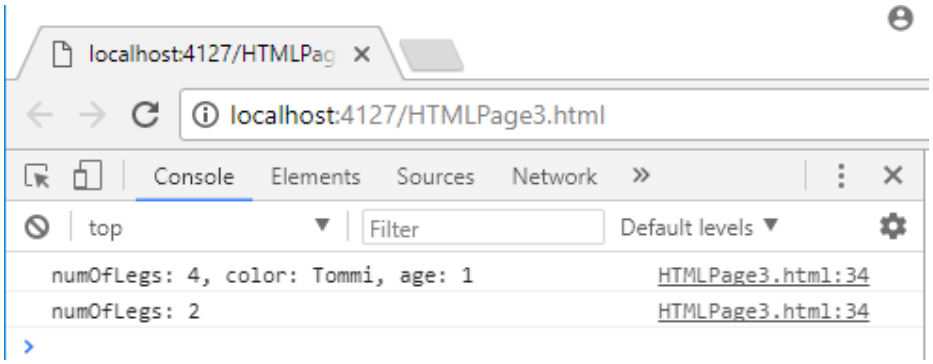


Polymorphism

בכדי לממש את עיקרון הפולימורפיזם, שהינו אחד מאבי היסוד בתכנות מונחה עצמים, נוכל ליצור function constructor של Animal ולהגדיר שCat יורש מ,Animal כאשר לשניהם יש את הפונקציה getInfo וכל אחד מבצע אותה במימוש המתאים לו, תוך שימוש במחלקת 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 בתוך פונקציה יש טווח הכרה לאורך הפונקציה (כולל בתוך פונקציות מקוננות), אבל הם לא קיימים מחוץ לפונקציה. המשתנים המוצהרים מחוץ לפונקציה הם משתנים גלובליים והם גלויים לאורך כל תוכנית., אולם נוכל בכל זאת ליצור משתנים מוסתרים בתוך בלוק קוד, על ידי הגדרת פונקציה פשוטה שתשמש כ- namespace זמני שבו ניתן להגדיר משתנים מבלי להוסיפם ל- global namespace

נניח, לדוגמה, שיש לנו מודול של קוד JavaScript וברצונינו להשתמש בו במספר תוכניות JavaScript שונות , ונניח שקוד זה מגדיר משתנים לאחסון תוצאות הביניים של החישוב שלו. הבעיה היא שמפני שהמודול הזה ישמש בתוכניות רבות ושונות, איננו יודעים אם המשתנים שהוא יוצר יתנגשו עם משתנים המשמשים את התוכניות המיובאות. הפתרון, כמובן, הוא לשים את הקוד לתוך פונקציה ולאחר מכן להפעיל את הפונקציה. בדרך זו, משתנים שהיו גלובלים 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 ההגדרה של המשתנה היחיד הזה ע"י הגדרת פונקציה אנונימית בביטוי אחד:

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

7.2. class

לפני 6ES, יצירתclass הייתה עניין מסובך. ב- 6ESניתן ליצור class באמצעות ה- keyword החדש class .

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

- 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 הקצאת זיכרון עבור אובייקטים של הכיתה.

- methods - **methods** מייצגות פעולות שאובייקט יכול לבצע. הם נכתבים ללא הקידומת של המילה השמורה function.

מרכיבים אלה ביחד מכונים 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 מייצג 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 – בניגוד למשתנים ולפונקציות, מחלקות לא מבצעות hoisting – ולא מוכרות מעל השורות בהן הן הוגדרו.

יצירת אובייקטים

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

```
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 לגשת למאפיינים ולפונקציות של המחלקה באמצעות שם האובייקט בתוספת סימון 'נקודה' ואז פניה לפונקציה. לדוגמה:

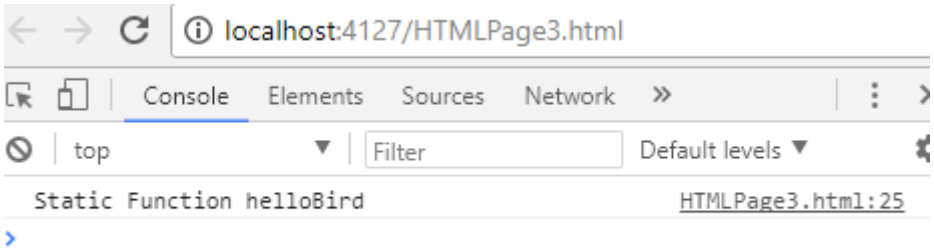
```
<!DOCTYPE html>

<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. מטודות סטטיות נגישות דרך שם המחלקה. לדוגמה:

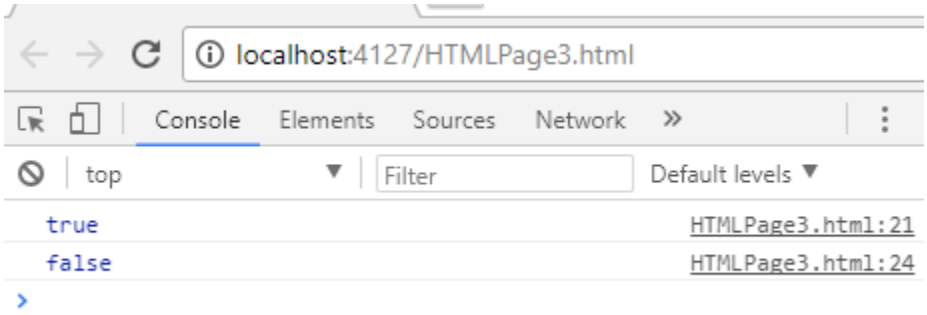
```
<!DOCTYPE html>

<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 שייך 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

6ES תומכת במושג הירושה. ירושה היא היכולת של תוכנית ליצור תבניות של ישויות חדשות מתבנית ישות קיימת - המחלקה הבסיסית שמשמשת מחלקות חדשות יותר נקראת מחלקת האב. והמחלקות החדשות שיורשות ממנה מכונות נגזרות.

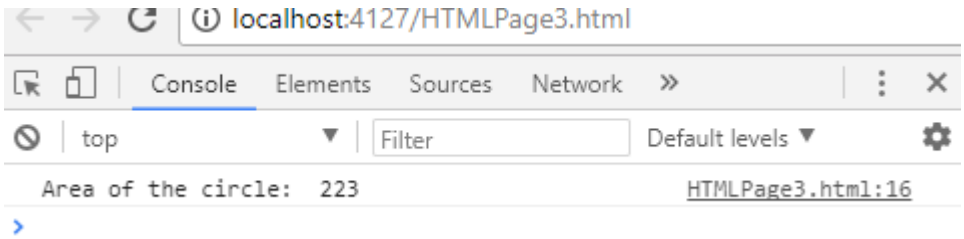
מחלקה אחרת יורשת ממחלקה אחרת באמצעות המילה השמורה extends. נגזרות יורשות את כל המאפיינים והשיטות, למעט בנאי ממחלקת האב.

להלן התחביר ליצירת מחלקה יורשת:

```
class child_class_name extends parent_class_name
```

דוגמה מלאה:

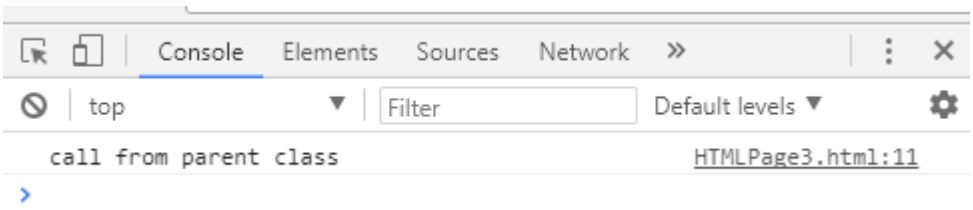
```
<!DOCTYPE html>

<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>
```



<u>ניתן לסווג את הירושה כ -</u>

- ירושה יחידה - כל מחלקה יכולה, לכל היותר, להאריך ממחלקת בסיס אחת
- ירושה מרובה - מחלקה יכולה לרשת בירוש מכמה מחלקות בסיס – יכולות זו לא מתאפשרת ב- 6ES . אינו תומך.
- ירושת **Multi-level**– לדוגמה:

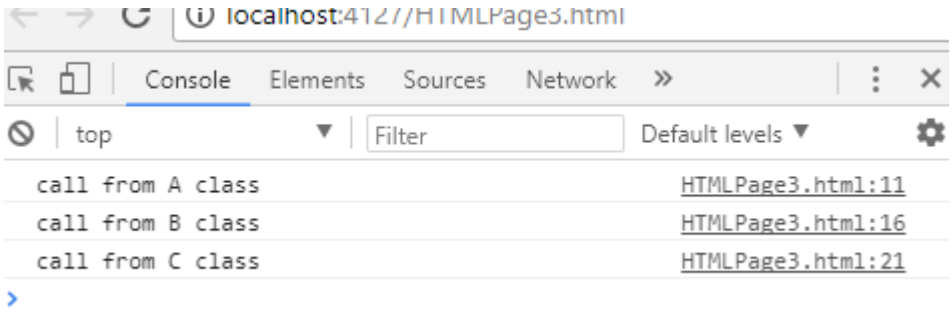
```
<!DOCTYPE html>

<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

Method Overriding הוא מצב שבו הנגזרת מגדירה 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>
```

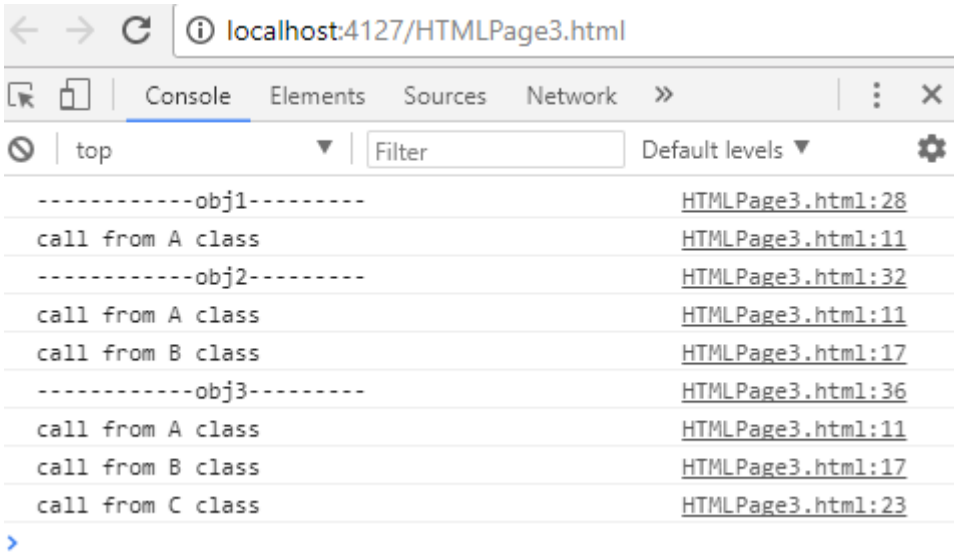


Super Keyword

6 מאפשר לנגזרת, להפעיל את המטודות של מחלקת הבסיס זה מושג באמצעות המילה השמורה ES המשמשת להפניה להורה הישיר של המחלקה. לדוגמה:super

```
<!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>
```



7.3. תרגילים

תרגילים בנושא constructor function & inheritance

תרגיל 1

1. צרו Constructor Function בשם Shape המתארת צורה כללית ומכילה:
   - x – מיקום הצורה על ציר x.
   - y – מיקום הצורה על ציר y.
   - color – צבע הצורה.

1. הוסיפו ל-Shape פונקציה המחזירה את מרחק הצורה מראשית הצירים.
   כלומר מרחק הנקודה (x,y) מראשית הצירים.
   נוסחה: שורש של (x בריבוע + y בריבוע).

1. בצעו דריסה של toString והחזירו ממנה מחרוזת המכילה את פרטי הצורה בפורמט הבא:

X = ____ , Y = ____ , Color = ____

תרגיל 2

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

2. בצעו דריסה של פונקציית ה-toString כך שהפעם היא תחזיר את פרטי העיגול בפורמט הבא:

X = ____ , Y = ____ , Color = ____ , Radius = ____

3. הוסיפו ל-Circle מאפיין סטטי בשם PI השווה ל-3.14.

4. הוסיפו ל-Circle פונקציה נוספת בשם getArea המחזירה את שטח הפנים של העיגול.

5. הוסיפו ל-Circle פונקציה נוספת בשם getPerimeter המחזירה את היקף העיגול.

6. השתמשו ב-Circle שבניתם 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

מכיל את המאפיינים:

- זיכרון מעבד (4-16)
- זיכרון דיסק (200-3000)

- דגם מעבד
- מחיר (800-20000)
- שנות אחריות (0-5)

מכיל את הפונקציות:

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

<u>מחשב נייח (יורש ממחשב)</u>

מכיל את המאפיינים:

- האם העכבר אלחוטי (בוליאני)
- גודל מסך מחשב נייח (11-18)

מכיל את הפונקציות:

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

<u>מחשב נייד (יורש ממחשב)</u>

מכיל את המאפיינים:

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

מכיל את הפונקציות:

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

1. צור פונקציה בשם **executeActions** המקבלת משתנה ומבצעת את כל הפונקציות שיש לאותו 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

הוסיפו לפרויקט 2 קבצי JavaScript בעלי השמות הבאים:

globals.js

site.js

צרו ב-index.html קישור לסקריפטים הללו:

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

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

(יש לשים לב שהקישור ל-globals.js נמצא מעל הקישור ל-site.js)

בקובץ ה-globals.js בצעו:

1. Self-Invoked Function העוטפת את כל תוכן הקובץ
2. "use strict"
3. Namespace בשם globals השייך ל-window
4. פונקציה בשם getCurrentTime המחזירה את השעה הנוכחית (מחזירה ע"י return, לא מציגה ע"י alert)

בקובץ ה-site.js בצעו:

1. Self-Invoked Function העוטפת את כל תוכן הקובץ
2. "use strict"
3. קיראו לפונקציה getCurrentTime שנמצאת ב-Namespace שיצרתם, והציגו את הערך המוחזר ממנה ע"י alert

הריצו את האתר ובידקו שאכן מוצגת השעה הנוכחית.