* 1. Self-invoke functions

Immediately Invoked Function Expressions -**IIFE** - 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);

}

}();

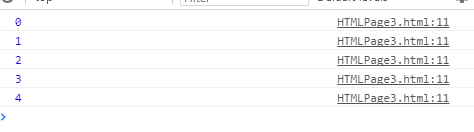
</script>

</head>

<body>

</body>

</html>



**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);

}

}());

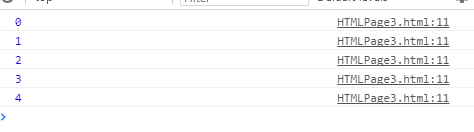
</script>

</head>

<body>

</body>

</html>



* 1. 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());

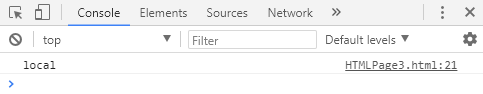
</script>

</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>

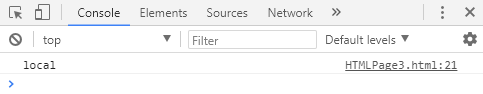
</head>

<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.

6. מחלקות הורשה ו prototype

1. 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: age ,color

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);

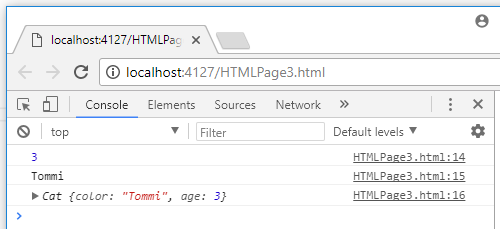
</script>

</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);

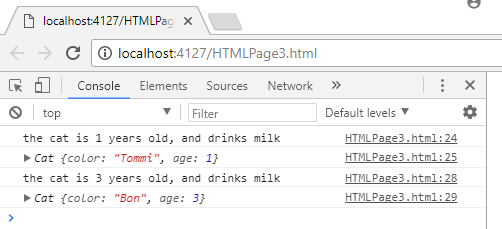
</script>

</head>

<body>

</body>

</html>



**יצירת מאפיינים ופונקציות סטטיות**

על ידי גישה לprotoype 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();

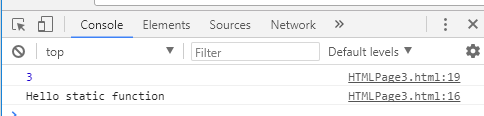
</script>

</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());

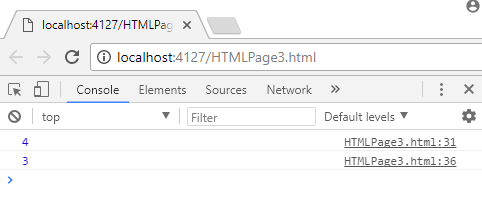
</script>

</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);

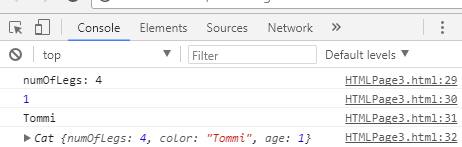
</script>

</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());

}

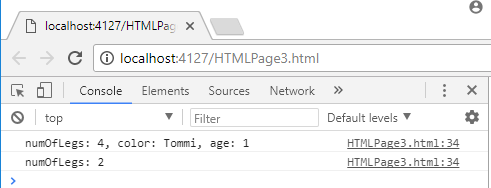
</script>

</head>

<body>

</body>

</html>

****

* 1. class

לפני ES6, יצירתclass הייתה עניין מסובך. ב- ES6ניתן 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

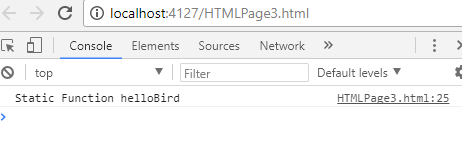
</script>

</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);

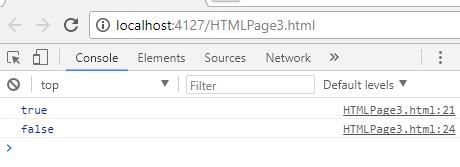
</script>

</head>

<body>

</body>

</html>

****

## Class Inheritance

ES6 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()

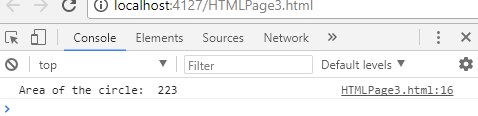
</script>

</head>

<body>

</body>

</html>



**ניתן לסווג את הירושה כ -**

* **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- . ES6 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

MethodOverridingהוא 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();

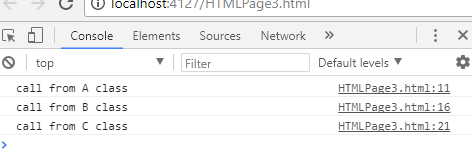
</script>

</head>

<body>

</body>

</html>



## Super Keyword

ES6 מאפשר לנגזרת, להפעיל את המטודות של מחלקת הבסיס זה מושג באמצעות המילה השמורה super המשמשת להפניה להורה הישיר של המחלקה.

לדוגמה:

<!DOCTYPE html>

<html lang="en" xmlns="http://www.w3.org/1999/xhtml">

<head>

<meta charset="utf-8" />

<title></title>

<script>

class A {

test() {

console.log("call from A class");

}

}

class B extends A {

test() {

super.test();

console.log("call from B class");

}

}

class C extends B {

test() {

super.test();

console.log("call from C class");

}

}

var obj1 = new A();

console.log("------------obj1---------");

obj1.test();

var obj2 = new B();

console.log("------------obj2---------");

obj2.test();

var obj3 = new C();

console.log("------------obj3---------");

obj3.test();

</script>

</head>

<body>

</body>

</html>

