Objects

כידוע, ישנם שמונה סוגי נתונים ב- JavaScript. שבעה מהם נקראים "פרימיטיביים", מכיוון שערכיהם מכילים דבר אחד בלבד (בין אם זה מחרוזת או מספר או כל דבר אחר).

לעומת זאת, אובייקטים משמשים לאחסון אוספי נתונים שונים וישויות מורכבות יותר.
ב- JavaScript אובייקטים חודרים כמעט לכל היבט בשפה.

ניתן ליצור אובייקט עם סוגריים מסולסלים {...} עם רשימת מאפיינים אופציונלית.
מאפיין הוא צמד "key: value", כאשר המפתח הוא מחרוזת (המכונה גם "property name"), והערך יכול להיות כל דבר.

אנו יכולים לדמיין אובייקט כארון עם מגירות. כל פיסת נתונים נשמרת במגירה על ידי שם מגירה = המפתח.
כך קל למצוא מגירה בשמה ולגשת למידע או להוסיף / להסיר מידע מתוכה.

ניתן ליצור אובייקט ריק ("ארון ריק") באמצעות אחד משני התחבירים:

```
let user = new Object(); // "object constructor" syntax
let user = {};  // "object literal" syntax
```

בד"כ משתמשים עם סוגריים מסולסלים – הכרזה זו נקראת אובייקט מילולי (object literal).

מילים ומאפיינים

אנו יכולים להכניס מאפיינים מיד ביצירת האובייקט ע"י key: value

```
let user = {      // an object
  name: "John",   // by key "name" store value "John"
  age: 30         // by key "age" store value 30
};
```

להסביר את הקוד – כל תו וכל הצהרה.

אנו יכולים להוסיף, להסיר ולקרוא מידע מהאובייקט בכל עת.
ניתן להגיע לערכי המאפיינים באמצעות סימון נקודה ובחירת מאפיין:

```
// get property values of the object:
alert( user.name ); // John
alert( user.age ); // 30
```

הערכים יכולים להיות מכל סוג שהם. בוליאני או מחרוזת לדוגמא:

```
user.isAdmin = true;
```

למחיקת מאפיין, ניתן להשתמש באופרטור delete

```
delete user.age;
```

אנחנו יכולים לכתוב מפתח של מאפיין מרובה מילים, אך נשים אותו בסוגריים:

```
let user = {
  name: "John",
  age: 30,
  "likes birds": true  // multiword property name must be quoted
};
```

אחרי כל מאפיין כולל המאפיין האחרון, נדרש לכתוב פסיק (,) .

סוגריים מרובעים

```
// this would give a syntax error
user.likes birds = true
```

עבור מאפיינים מרובי-מילים, הגישה לנקודה אינה פועלת:

JavaScript לא מבין זאת. הוא חושב שאנו פונים ל- user.likes ואז נותן שגיאת .

הנקודה מחייבת שהמפתח לא מכיל רווחים, לא מתחיל בספרה ולא כולל תווים מיוחדים ($ ו- _ מותרים).

פתרון לכך הוא שימוש בסוגריים מרובעים:

```
let user = {};

// set
user["likes birds"] = true;

// get
alert(user["likes birds"]); // true

// delete
delete user["likes birds"];
```

ניתן שערך המפתח ישתנה בזמן ריצה או יהיה תלוי בקלט המשתמש.

כך אנו משתמשים בו כדי לגשת למאפיין. זה נותן לנו גמישות במידה רבה:

```
let user = {
  name: "John",
  age: 30
};

let key = prompt("What do you want to know about the user?",
"name");

// access by variable
alert( user[key] ); // John (if enter "name")
```

אבל שיטת הנקודה לא תעבוד במקרה הזה

```
let key = "name";
alert( user.key ) // undefined
```

מאפיינים מחושבים (computed)

אנו יכולים להשתמש בסוגריים מרובעים באובייקט מילולי, בעת יצירת אובייקט.

זה נקרא מאפיינים מחושבים.

```
let fruit = prompt("Which fruit to buy?", "apple");

let bag = {
  [fruit]: 5, // the name of the property is taken from the variable
fruit
};

alert( bag.apple ); // 5 if fruit="apple"
```

להסביר מה קורה פה בקוד ולהראות דוגמא נוספת:

```
let fruit = prompt("Which fruit to buy?", "apple");
let bag = {};

// take property name from the fruit variable
bag[fruit] = 5;
```

סוגריים מרובעים חזקים בהרבה מסימון הנקודה. מצד אחד מאפשרים להגדיר כל שם למאפיינים, ומצד שני מסורבלים יותר לכתיבה. אז לרוב, כששמות המפתחות ידועים ופשוטים, משתמשים בנקודה ואם אנו זקוקים למשהו מורכב יותר, אנו עוברים לסוגריים מרובעים.

קיצור ערכי מאפיינים

בקוד אמיתי אנו משתמשים לעיתים קרובות במשתנים קיימים כערכים עבור שמות מאפיינים.

```
function makeUser(name, age) {
  return {
    name: name,
    age: age,
    // ...other properties
  };
}

let user = makeUser("John", 30);
alert(user.name); // John
```

בדוגמה לעיל, למאפיינים יש אותם שמות כמו משתנים.

מקרה של שימוש במשתנה למאפיין ולערך הוא כה נפוץ, עד כי יש קיצור שנועד לכך.

```
function makeUser(name, age) {
  return {
    name, // same as name: name
    age,  // same as age: age
    // ...
  };
}
```

ניתן להשתמש גם בצורה חלקית :

```
let user = {
  name,  // same as name:name
  age: 30
};
```

מגבלות על שמות מאפיינים

בניגוד למשתנים, ניתן להשתמש בשמות מאפיינים במילים שמורות כגון : return, let, for

ובמספרים , לדוגמא 0 שיומר וישמר "0" .

בדיקה האם קיים מאפיין, "operator "in

יתרון בולט של אובייקטים ב- JavaScript, בהשוואה לשפות רבות אחרות, הוא שאפשר לגשת לכל מאפיין. לא תהיה שום שגיאה אם המאפיין לא קיים.

קריאת מאפיין שאינו קיים פשוט מחזירה undefined:

```
let user = {};

alert( user.noSuchProperty === undefined ); // true means "no such property"
```

```
let user = { name: "John", age: 30 };

alert( "age" in user ); // true, user.age exists
alert( "blabla" in user ); // false, user.blabla doesn't exist
```

לולאת ה- "for... in"

כדי לעבור על כל מפתחות האובייקט, ניתן לבצע : for..in

```
let user = {
  name: "John",
  age: 30,
  isAdmin: true
};

for (let key in user) {
  // keys
  alert( key );  // name, age, isAdmin
  // values for the keys
  alert( user[key] ); // John, 30, true
}
```

שימו לב שבמבנה ה- "for" אנו מכריזים על משתנה לולאה שבכל לופ (איטרציה) מתבצע השמת המפתח למשתנה הזה (key).

Objects Referencing and copying

אחד ההבדלים המהותיים בין אובייקטים לעומת טיפוסים פרימיטיביים הוא שהאובייקטים מאוחסנים ומועתקים "by reference", ואילו טיפוסים פרימיטיביים: מחרוזות, מספרים, בוליאנים וכו '- מועתקים תמיד "by value".

נתחיל בפרימיטיבי, כמו מחרוזת:

```
let message = "Hello!";
let phrase = message;
```

כתוצאה מכך יש לנו שני משתנים עצמאיים, בכל אחד מאוחסן המחרוזת "hello!".

לעומת זאת, אובייקטים אינם כאלה.

משתנה המוקצה לאובייקט לא מאחסן את האובייקט עצמו, אלא את "הכתובת בזיכרון" - במילים אחרות "הפניה – reference" אליו.

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

האובייקט מאוחסן איפשהו בזיכרון , ואילו למשתנה user יש "reference" אליו.

אנו עשויים לחשוב על משתנה של אובייקט, כמו למשל user, כמו דף נייר עם כתובת האובייקט עליו.

כאשר אנו מבצעים פעולות עם האובייקט, למשל user.name, מנוע JavaScript מסתכל על מה שנמצא בכתובת זו ומבצע את הפעולה על האובייקט בפועל.

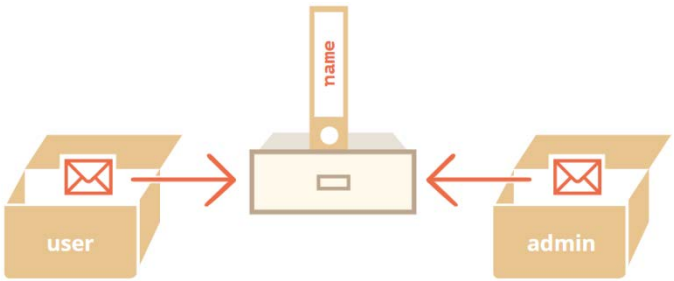
ולמה זה חשוב?

כאשר משתנה אובייקט מועתק, ההפניה מועתקת, אך האובייקט עצמו אינו משוכפל.

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

let admin = user; // copy the reference
```

כעת יש לנו שני משתנים שכל אחד מהם מאחסן **הפנייה** לאותו אובייקט.



כפי שניתן לראות, עדיין יש אובייקט אחד, אך כעת עם שני משתנים המופנים אליו.

אנו יכולים להשתמש בכל אחד מהמשתנים כדי לגשת לאובייקט ולשנות את תוכנו:

```
let user = { name: 'John' };

let admin = user;

admin.name = 'Pete'; // changed by the "admin" reference

alert(user.name); // 'Pete', changes are seen from the "user"
reference
```

להסביר את הקוד הנ"ל.

Comparison by reference - השוואה לפי התייחסות

שני אובייקטים שווים רק אם הם אותו אובייקט.
למשל, כאן a ו- b מתייחסים לאותו אובייקט, ולכן הם שווים:

```
let a = {};
let b = a; // copy the reference

alert( a == b ); // true, both variables reference the same object
alert( a === b ); // true
```

וכאן שני אובייקטים עצמאיים אינם שווים, למרות שהם נראים זהים (שניהם ריקים):

```
let a = {};
let b = {}; // two independent objects

alert( a == b ); // false
```

Object.assign - שכפול ומיזוג

העתקת משתנה של אובייקט יוצרת הפנייה נוספת לאותו אובייקט.
אבל מה אם נצטרך לשכפל אובייקט? ליצור עותק עצמאי, שיבוט?

זה גם בר ביצוע, אבל קצת יותר קשה, מכיוון שאין שום שיטה מובנית לכך ב- JavaScript.
אך לעיתים רחוקות יש צורך - העתקה על ידי הפניה טובה לרוב.

אבל אם אנחנו באמת נרצה להעתיק אובייקט, עלינו ליצור אובייקט חדש ולשכפל את המבנה של הקיים על ידי איטרציה על כל מאפייניו והעתקתם ברמה הפרימיטיבית.

ככה:

```
let user = {
  name: "John",
  age: 30
};

let clone = {}; // the new empty object

// let's copy all user properties into it
for (let key in user) {
  clone[key] = user[key];
}
```

```
// now clone is a fully independent object with the same content
clone.name = "Pete"; // changed the data in it

alert( user.name ); // still John in the original object
```

כמו כן אנו יכולים להשתמש בשיטה Object.assign לשם כך.

```
Object.assign(dest, [src1, src2, src3...])
```

הפרמטר הראשון dest הוא אובייקט היעד – אליו ישוכפלו המאפיינים.
הפרמטרים הנוספים src1, ..., srcN (יכולים להיות רבים ככל שיידרש) הם אובייקטים מקוריים.
הפונקציה מעתיקה את המאפיינים של כל אובייקטי המקור src1, ..., srcN אל היעד (dest).
במילים אחרות, מאפייני כל הארגומנטים החל מהארגומנט השני מועתקים לאובייקט הראשון.
אובייקט dest חוזר מהפונקציה.

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

let permissions1 = { canView: true };
let permissions2 = { canEdit: true };

// copies all properties from permissions1 and permissions2 into
user
Object.assign(user, permissions1, permissions2);

// now user = { name: "John", canView: true, canEdit: true }
```

אם שם המאפיין המועתק כבר קיים, תתבצע החלפה:

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

Object.assign(user, { name: "Pete" });

alert(user.name); // now user = { name: "Pete" }
```

נוכל להשתמש ב-Object.assign על מנת להחליף את for…in בביצוע ההעתקה פשוטה. (*תרגיל*)

```
let user = {
  name: "John",
  age: 30
};

let clone = Object.assign({}, user);
```

nested cloning (שכפול מקונן)

מאפיינים יכולים להיות הפניות לאובייקטים אחרים.

```
let user = {
  name: "John",
  sizes: {
    height: 182,
    width: 50
  }
};
```

```
alert( user.sizes.height ); // 182
```

הפתרון לשכפול מלא של המאפיין sizes שהוא גם אובייקט , ככה:

```
let user = {
  name: "John",
  sizes: {
    height: 182,
    width: 50
  }
};

let clone = Object.assign({}, user);

alert( user.sizes === clone.sizes ); // true, same object

// user and clone share sizes
user.sizes.width++;       // change a property from one place
alert(clone.sizes.width); // 51, see the result from the other one
```

כדי לתקן את זה, עלינו להשתמש בלולאת שכפול שבודקת כל ערך של user[key], ואם זה אובייקט, אז לשכפל גם את המבנה שלו. זה נקרא "שכפול עמוק - deep cloning".
אנו יכולים להשתמש ברקורסיה כדי ליישם אותה.

Garbage Collection – איסוף זבל

ניהול זיכרון ב- JavaScript מתבצע באופן אוטומטי ובלתי נראה לנו.
אנו יוצרים משתנים פרימיטיביים, אובייקטים, פונקציות ... כל אלו צורכים זיכרון.

מה קורה כשלא צורכים יותר את המשתנים האלו ? איך מנוע JavaScript מגלה אותם ומנקה אותם?

נגישות
הרעיון העיקרי של ניהול זיכרון ב- JavaScript הוא הנגישות.
במילים פשוטות, ערכים "נגישים" הם נגישים או שמישים איכשהו. הם מובטחים להיות מאוחסנים בזיכרון.

יש מערך בסיס של ערכים הנגישים מטבעם, שלא ניתן למחוק אותם מסיבות ברורות.

לדוגמה:

הפונקציה המבצעת כעת, המשתנים והפרמטרים המקומיים שלה.
פונקציות אחרות בשרשרת השיחות המקוננות הנוכחיות, המשתנים והפרמטרים המקומיים שלהם.
משתנים גלובליים.
(יש גם כמה אחרים, פנימיים)
ערכים אלה נקראים שורשים.

כל ערך אחר נחשב נגיש אם ניתן להגיע אליו משורש באמצעות הפניה או על ידי שרשרת הפניות.

למשל, אם יש אובייקט במשתנה גלובלי, ולאובייקט הזה יש מאפיין שמפנה לאובייקט אחר, אובייקט זה נחשב נגיש. ואלו שאליהם הוא מפנה ניתן להגיע גם. דוגמאות מפורטות לעקוב.

יש תהליך רקע במנוע ה- JavaScript שנקרא אספן אשפה. הוא עוקב אחר כל האובייקטים ומסיר את אלה שהפכו לבלתי נגישים.

Object methods

אובייקטים נוצרים בדרך כלל כדי לייצג ישויות מהעולם האמיתי, כמו משתמשים, הזמנות וכן הלאה:

```
let user = {
  name: "John",
  age: 30
};
```

ובעולם האמיתי, משתמש יכול לבצע פעולות מסוימות, כמו למשל לבחור משהו מסל הקניות, להתחבר, להתנתק וכו'.
פעולות מיוצגות ב- JavaScript על ידי פונקציות במאפיינים.

```
let user = {
  name: "John",
  age: 30
};

user.sayHi = function() {
  alert("Hello!");
};

user.sayHi(); // Hello!
```

כאן השתמשנו ב- function expression כדי ליצור פונקציה ולהקצות אותה למאפיין user.SayHi של האובייקט.
כך נוכל לקרוא לפונקציה user.sayHi() ופתאום המשתמש יכול לדבר עכשיו :]

פונקציה שהיא מאפיין של אובייקט נקראת method – שיטה.

אז באובייקט user יש לנו method שנקרא sayHi.
כמובן שנוכל להשתמש בפונקציה שהוכרזה מראש כ- method, כך:

```
let user = {
  // ...
};

// first, declare
function sayHi() {
  alert("Hello!");
};

// then add as a method
user.sayHi = sayHi;

user.sayHi(); // Hello!
```

יש תחביר קצר יותר לשיטות באובייקט מילולי:

```
// these objects do the same

user = {
  sayHi: function() {
    alert("Hello");
  }
};

// method shorthand looks better, right?
user = {
  sayHi() { // same as "sayHi: function(){...}"
    alert("Hello");
  }
};
```

"this" in methods

מקובל ש-method של אובייקט צריכה לגשת למידע המאוחסן באובייקט עצמו כדי לבצע את עבודתה.

לדוגמה, ייתכן שהקוד בתוך ()user.sayHi זקוק לשם המשתמש.

כדי לגשת לאובייקט עצמו, method נדרש להשתמש במילת המפתח **this**.
ניתן לחשוב ש - this הוא האובייקט "לפני נקודה", והאובייקט הוא זה שקורא ל-method/פונקציה.

```
let user = {
  name: "John",
  age: 30,

  sayHi() {
    // "this" is the "current object"
    alert(this.name);
   }

};

user.sayHi(); // John
```

ניתן לבצע ככה את sayHi:

```
    alert(user.name); // "user" instead of "this"
```

אבל קוד כזה הוא לא אמין. במידה ונעתיק את האובייקט user לאובייקט אחר. השם שיוצג יהיה אחר.

```
let user = {
  name: "John",
  age: 30,

  sayHi() {
    alert( user.name ); // leads to an error
  }
```

```
};


let admin = user;
user = null; // overwrite to make things obvious

admin.sayHi(); // TypeError: Cannot read property 'name' of null
```

"this" לא מחויב/מקושר

ב- JavaScript, מילת מפתח this מתנהגת בשונה לרוב שפות התכנות האחרות.

ניתן להשתמש בה **בכל** פונקציה, לא רק בשיטה של אובייקט.

```
function sayHi() {
  alert( this.name );
}
```

הערך של this מחושב במהלך זמן הריצה, בהתאם להקשר :

```
let user = { name: "John" };
let admin = { name: "Admin" };

function sayHi() {
  alert( this.name );
}

// use the same function in two objects
user.f = sayHi;
admin.f = sayHi;

// these calls have different this
// "this" inside the function is the object "before the dot"
user.f(); // John  (this == user)
admin.f(); // Admin  (this == admin)

admin['f']();  // Admin (dot or square brackets access the method – doesn't matter)
```

הכלל פשוט: אם נקרא obj.f(), אז this יהיה obj במהלך הקריאה של f.

שימו לב - Calling without an object: this == undefined

```
function sayHi() {
  alert(this);
}

sayHi(); // undefined
```

כלל נוסף : **Arrow functions have no "this"**

this בפונקציית חץ יילקח מהפונקציה החיצונית "הרגילה".