Javascript II: Objects and Arrays

Pau Fernández Rafa Genés Jose L. Muñoz

Universitat Politècnica de Catalunya (UPC)

Table of Contents

Objects

Objects

An object is a **table** (or map), associating *properties* (strings) with values.

Objects are typically created using literals and given some initial properties:

```
let emptyObj = {};
let person = {
  last: "Bond",
  first: "James",
  age: 27
};
```

To access properties we use the dot:

```
person.last
person.age
```

Deleting Properties

You can remove a property on an object by using **delete**:

```
let character = {
  name: "Han Solo",
  age: 27
};

delete character.age;
```

Properties with general names

Property names in an object are **general strings** (with spaces, symbols, etc.).

These properties are accessed with brackets **[]**.

```
let obj1 = {};
obj1[' like '] = true;
obj1['/* awesome */'] = 42;
obj1[0] = 'ZERO!'; // 0 converts to '0'!
var obj2 = {
   ['my prop with spaces']: 1,
};
console.log(obj1, obj2);
```

The output is

```
{ '0': 'ZERO!', ' like ': true, '/* awesome */': 42 }
{ 'my prop with spaces': 1 }
```

Checking if a property exists

If you access a property, **there will be no error even if the property doesn't exist** (it returns **undefined**).

How do you test for the existence of a certain property?

- Using special syntax, the **in** operator:

```
if ("prop" in object) {
  // property present (possibly inherited)
}
```

- Using the **hasOwnProperty** method:

```
if (object.hasOwnProperty("prop")) {
  // property present in the object itself
}
```

Iterating Properties

A special syntax allows iteration over all properties of an object:

```
for (let property in object) {
  // execute for every property (key) in the object
}
```

The following code:

```
let animal = { name: "Rufus", weight: 21.5 };
for (let prop in animal) {
  console.log(prop, animal[prop]);
}
```

produces

```
name Rufus
weight 21.5
```

keys, values, entries

`Object.keys` → Return object properties as a list.
`Object.values` → Return associated values as a list.
`Object.entries` → Return all pairs [key, value] as a list.

```
let obj = {
  a: 1,
  b: 'hi',
  c: true
};

console.log(Object.keys(obj));   // -> ['a', 'b', 'c']
console.log(Object.values(obj)); // -> [1, 'hi', true]
console.log(Object.entries(obj));
// -> [['a', 1], ['b', 'hi'], ['c', true]]
```

Initialization with Analogous Variables

If we have some variables

```
let name = 'James';
let age = 27;
```

we can create an object with fields taken from those variables

```
let obj = {
  name: name,
  age: age
};
```

When the fields coincide with the variables, we can omit them

```
let obj = { name, age };
```

Table of Contents

JSON

JavaScript Object Notation (JSON)

JSON is a standard format for information exchange inspired in the object notation in Javascript:

```
{
   "number field": 1, "null": null,
   "string_field": "the string",
   "boolean (positive)": true, "(negative)": false,
   "object": {
      "subfield1": "enough"
   },
   "array_field": [
      "a string", 1001, { "another": "object" }
   ]
}
```

Details: json.org

JSON.stringify and JSON.parse

JSON.stringify: convert Javascript objects to a JSON string:

```
let obj = { a: 1, b: [2, 3] };
console.log(JSON.stringify(obj));  // {"a":1,"b":[2,3]}
```

JSON.parse: convert a JSON string into Javascript (in-memory) objects:

```
let json = JSON.parse(`{ "d": 4, "e": [5, 6] }`);
console.log(JSON.parse(json));  // { d: 4, e: [ 5, 6 ] }
```

Indentation in `JSON.stringify`

The 3rd parameter in `JSON.stringify` is the indentation:
a) number of spaces, or
b) the string to use (such as `'\t'`).

```
JSON.stringify({ a: 1, b: 'hi', c: true }, null, 2)
```

```
{
  "a": 1,
  "b": "hi",
  "c": true
}
```

Table of Contents

Destructuring

Destructuring

Given this object

```
let message = {
  from: "James Bond", to: "M", text: "Target terminated"
};
```

accessing its fields can be cumbersome

```
console.log(
 `From: ${message.from}, To: ${message.to}, Text: ${message.text}`
);
```

We can ”destructure” an object with a special let syntax:

```
let { from, to, text } = message;
console.log(`From: ${from}, To: ${to}, Text: ${text}`);
```

Destructuring: Mapping to New Names

Destructuring expressions also let us change variables names:

```
let header = {
  title: "Pastries",
  width: 121, height: 83,
  border: true,
};
```

We can map `width` to `w`, `height` to `h`:

```
const { width: w, height: h, title } = header;
console.log("Title: " + title);
console.log("Width: " + w);
console.log("Height: " + h);
```

Destructuring: Nested Objects

Given a nested object:

```
let header = {
  title: "Pastries",
  size: { width: 121, height: 83 },
  border: true,
}
```

Desctructuring can be also nested:

```
const { title, size: { width: w, height: h } } = header;
console.log("Title: " + title);
console.log("Width: " + w);
console.log("Height: " + h);
```

Desctructuring: Spread

The spread operator can be used to destructure ”the rest” of properties:

```
let circle = {
    x: 1,
    y: 2,
    radius: 10.0,
    color: "blue"
};
let { x, y, ...rest } = circle;

console.log(x);    // -> 1
console.log(y);    // -> 2
console.log(rest); // ->  radius: 10.0, color: "blue"
```

Destructuring parameters

Destructuring is also allowed in function parameters:

```
function fullname({ first, last }) {
  console.log(`${first} ${last}`);
}

let person = {
  first: 'Ryan',
  last: 'Dahl'
};

fullname(person);
```

Table of Contents

Arrays

Declaring arrays

There are two forms of array declaration: creating them like objects, or as array literals:

```
let empty1 = new Array();
let empty2 = [];
let marxes = ["Groucho", "Chicco", "Harpo"];
let primes = [2, 3, 5, 7, 11, 13, 17];
let soup = [1, true, null, 'hi', NaN];
```

The `length` is a special property that always contains the number of elements of an array.

```
console.log(empty1.length);  // -> 0
console.log(marxes.length);  // -> 3
console.log(soup.length);    // -> 5
```

Push

To add an element to the end of an array, use **push**:

```
let A = [];
A.push(2);
A.push(3);
A.push(5);
console.log(A);  // -> [2, 3, 5]
```

Pop

To remove an element from the end, use **pop**:

```
let A = [2, 3, 5, 7, 11, 13];
let p = A.pop();
let q = A.pop();
console.log(p);  // -> 13
console.log(q);  // -> 11
console.log(A);  // -> [2, 3, 5, 7]
```

push and *pop* are *fast*, *because array elements don't move*

Shift

To remove the first element of an array, use **shift**:

```
let A = [2, 3, 5, 7, 11, 13];
let p = A.shift();
let q = A.shift();
console.log(p);  // -> 2
console.log(q);  // -> 3
console.log(A);  // -> [5, 7, 11, 13]
```

Unshift

To add an element at the beginning of the array, use **unshift**:

```
let A = [5, 7, 11, 13];
A.unshift(3);
A.unshift(2);
console.log(A);  // -> [2, 3, 5, 7, 11, 13]
```

shift and *unshift* are **slow**, *because they move all elements.*

Distinguishing objects from arrays

Arrays do not form a separate language type. They are based on objects.

`typeof` does not help to distinguish an object from an array

```
console.log(typeof {});  // -> object
console.log(typeof []);  // -> object
```

To know if an object is an array, use `Array.isArray`

```
console.log(Array.isArray({}));  // -> false
console.log(Array.isArray([]));  // -> true
```

Iterating arrays

A typical way of iterating over all elements of an array is the classic `for` using indices:

```
let A = [2, 3, 5, 7, 11, 13];
for (let i = 0; i < A.length; i++) {
  console.log(A[i]);
}
```

But the `for..of` syntax allows us do the same more succintly:

```
let A = [2, 3, 5, 7, 11, 13];
for (let val of A) {
  console.log(val);
}
```

This syntax does not provide the index, so it is not suited for loops where you need it.

Deleting elements from arrays

Deleting an element from an array is like deleting a property from an object:

```
let A = [1, 2, 3, 4, 5];
delete A[2];
console.log(A);     // -> [ 1, 2, <1 empty item>, 4, 5 ]
console.log(A[2]);  // -> undefined
```

Slice

```
<Array>.slice([begin [, end]])
```

`slice` returns part of an array (or all) **without modifying** the original array. We can specify the start and end position (end is not included).

```
let message = "Unbelievable";
let numbers = [2, 3, 5, 7, 11, 13];
console.log(message.slice(2, 8) + "e");  // -> believe
console.log(numbers.slice(1, 4));  // -> [3, 5, 7]
```

If you omit the end position it means ”until the end”:

```
console.log(message.slice(2));  // -> believable
console.log(numbers.slice(4));  // -> [ 11, 13 ]
```

Omitting both start and end copies the whole array.

Splice

```
<Array>.splice(index[, deleteCount [, elem1, ..., elemN]])
```

This method is used to **modify** an array (add, remove and insert elements).

Deletion

```
let A = [1, 2, 3, 4, 5, 6];
A.splice(2, 2);  // -> [1, 2, 5, 6]
```

Replacement

```
let A = [1, 2, 3, 4, 5, 6];
A.splice(2, 2, 30, 40);  // -> [1, 2, 30, 40, 5, 6]
```

Insertion

```
A = [1, 2, 3, 4, 5, 6];
A.splice(2, 0, 2.4, 2.6); // -> [1, 2, 2.4, 2.6, 3, 4, 5, 6]
```

concat

```
<Array>.concat(arg1, arg2...) // -> <Array>
```

Concatenate all the arguments (if they are arrays add all the elements) and returns a new array:

```
let A = [1, 2];
A = A.concat(3, 4);
A = A.concat([5, 6, 7]);
A = A.concat([8, 9], 10, [11, 12, 13]);

console.log(A); // -> [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]
```

indexOf and lastIndexOf and includes

```
<Array>.indexOf(value, from) // -> <Number>
<Array>.lastIndexOf(value, from) // -> <Number>
<Array>.includes(value, from) // -> <Boolean>
```

`indexOf` looks for `value` starting at position `from`.

`lastIndexOf` is analogous to `indexOf` but searches in reverse (from the end).

`includes` behaves as `indexOf` but only returns `true` if found, not the position.

```
let A = [1, 0, false, 0, 1, false];
console.log(A.indexOf(0));          // 1
console.log(A.indexOf(false));      // 2
console.log(A.indexOf(null));       // -1
console.log(A.lastIndexOf(0));      // 3
console.log(A.lastIndexOf(false)); // 5
console.log(A.includes(1));         // true
```

The three methods use the === operator for comparisons.

reverse

```
<Array>.reverse()
```

The **reverse** simply reverses the order of the elements in an array:

```
let A = [1, 2, 3, 4, 5];
A.reverse();
console.log(A);  // -> [5, 4, 3, 2, 1]
```

`split` and `join`

```
<String>.split(separator) // -> <Array>
<Array>.join(separator) // -> <String>
```

`split` and `join` are complementary methods for strings and arrays, respectively.

`split` creates an array of the parts that were separated by `separator` in a string.

```
let group = 'Groucho, Chicco, Harpo';
let names = group.split(', ');
console.log(names); // -> ['Groucho', 'Chico', 'Harpo']
```

`join` joins the elements of an array back into a string:

```
let elems = ['red', 'green', 'blue'];
let colors = elems.join(' : ');
console.log(colors); // -> 'red : green : blue'
```

Spread operator in arrays

The spread operator allows us to splice an array into another in any position:

```
var elems = ['two', 'three'];
var whole = ['one', ...elems, 'four', 'five'];
console.log(whole);  // ["one", "two", "three", "four", "five"]
```

It is useful also to **copy** arrays:

```
var array = [1, 2, 3, 4, 5];
var copy  = [...array];
```

And to **concatenate** them:

```
var a1 = [1, 2];
var a2 = [3, 4, 5];
var a3 = [...a1, ...a2];

console.log(a3);  // -> [1, 2, 3, 4, 5]
```

Spreading Parameters

The spread operator can be used to spread parameters:

```
function f(a, b, c) {
  return a + b + c;
}

let array1 = [-1, 3, 7];
let array2 = [0, -3];
console.log(f(...array1));    // -> 9
console.log(f(3, ...array2)); // -> 0
console.log(f(5, ...array1)); // -> 7
```

Table of Contents

Array Functional Methods

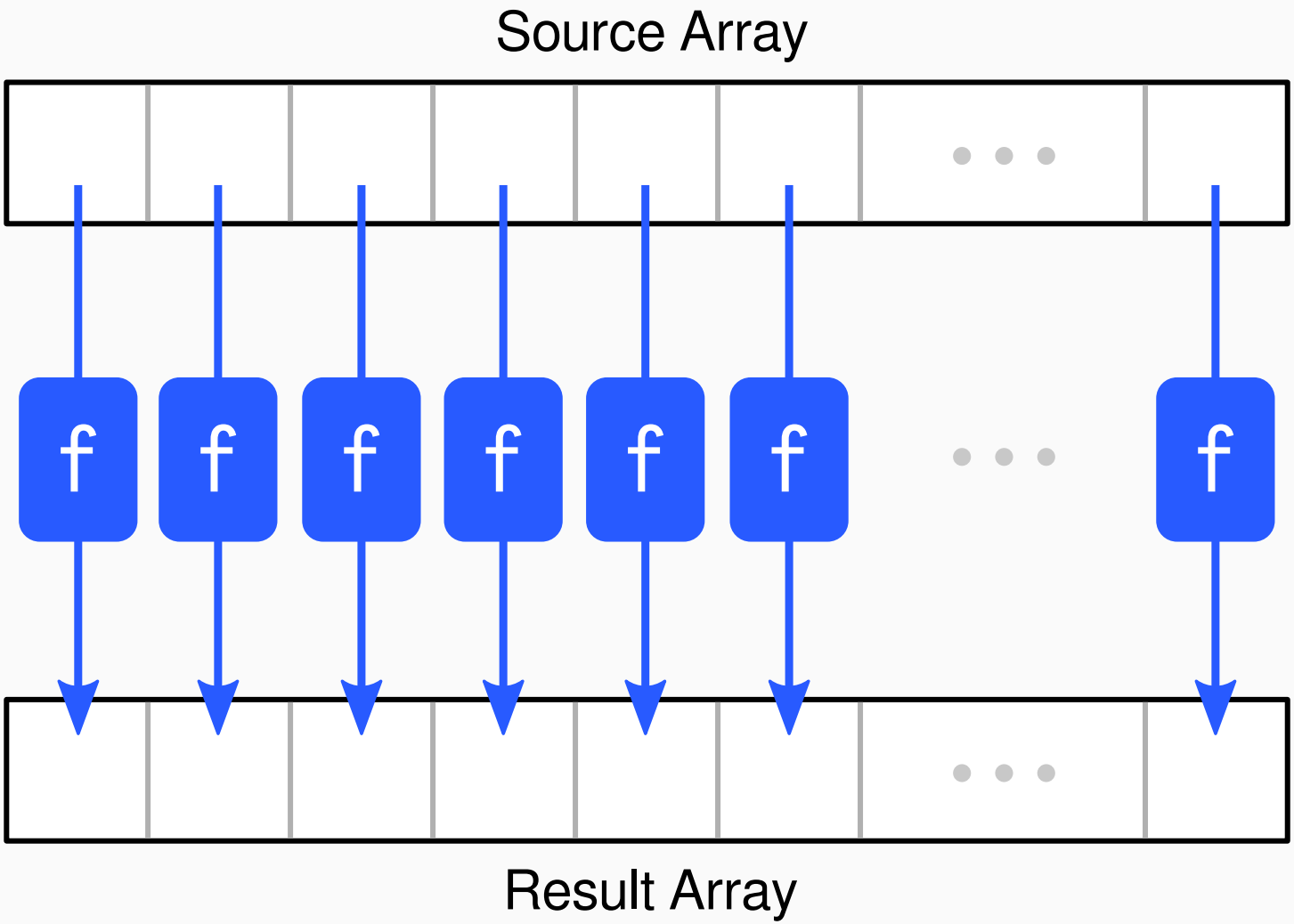
```
let results = <Array>.map((item, index, array) => {
  // returns the new value instead of item
})
```

The method **map** applies the function to each element and replaces that element with the result given by the function.

```
let A = [1, 2, 3, 4, 5]
console.log(A.map(n => n * 2)) // -> [2, 4, 6, 8, 10]
console.log(A.map(n => 'o'.repeat(n)))
// -> ['o', 'oo', 'ooo', 'oooo', 'ooooo']

let figures = [
   {shape: 'rect', size: 5},
   {shape: 'circle', size: 10},
]
console.log(figures.map(f => f.shape)) // -> ['rect', 'circle']
```

`map` diagram



The `map` Algorithm

The `map` algorithm could be implemented as follows:

```
function map(array, fn) {
  let result = []
  for (let i = 0; i < array.length; i++) {
    result.push(fn(array[i], i, array))
  }
  return result
}
```

forEach

```
<Array>.forEach((item, index, array) => {
  // Do something with item
})
```

The `forEach` method just executes a function for every element in an array (but not producing a new array).

```
let result = []
["be", "to", "not", "or", "be", "to"].forEach(item => {
  result.unshift(item)
})
console.log(result.join(' '))
```

filter

```
let results = <Array>.filter((item, index, array) => {
  // should return true if the item passes the filter
})
```

The `filter` method repetitively calls the function for every element in the original array and returns a new array with the elements in which the function returned `true`.

```
let students = [
   { name: 'Peter', mark: 6.7 },
   { name: 'Paul',  mark: 3.9 },
   { name: 'Mary',  mark: 9.2 },
]
let passed = students.filter((student) => student.mark > 5.0)
passed.forEach(s => console.log(s.name))
// -> Peter
// -> Mary
```

The `filter` Algorithm

The `filter` algorithm could be implemented as follows:

```
function filter(array, fn) {
  let result = []
  for (let i = 0; i < array.length; i++) {
    if (fn(array[i], i, array)) {
      result.push(array[i])
    }
  }
  return result
}
```

find and **findIndex**

```
let item = <Array>.find((item, index, array) => {
  // should return true if the item is what we are looking for
})
let index = <Array>.findIndex((item, index, array) => { ... })
```

The function is repetitively called for every element in the array (**item** being the element, **index** the position and **array** the array itself).

If the function returns **true**, **find** returns the element. If the condition is not met, **find** returns **undefined**.

findIndex behaves like **find** but returns the index of the element instead of the element itself.

```
console.log([3, 5, -8, 9, 2].find(x => x < 0))      // -> -8
console.log([3, 5, -8, 9, 2].findIndex(x => x < 0)) // -> 2
```

sort

```
<Array>.sort((a, b) => {
  // return zero if:              a === b
  // return positive number if:  a greater than b
  // return negative number if:  a less than b
})
```

Sorts the elements of an array *in place* and returns the array. The default sort order *converts elements to strings*, then compares.

```
let fruits = [
  { name: "orange", eur: 2.5 },
  { name: "kiwi", eur: 5.5 },
  { name: "apple", eur: 1.75 },
]
let names = fruits.map(f => f.name)
names.sort()                          // sort the names
fruits.sort((a, b) => b.eur - a.eur) // sort by price (desc)
```

reduce and reduceRight

```
<Array>.reduce((accum, item, index, array) => {
  // return the new value of the accumulator
}, initialValue)
<Array>.reduceRight((acc, it, i, ar) => { ... }, init)
```

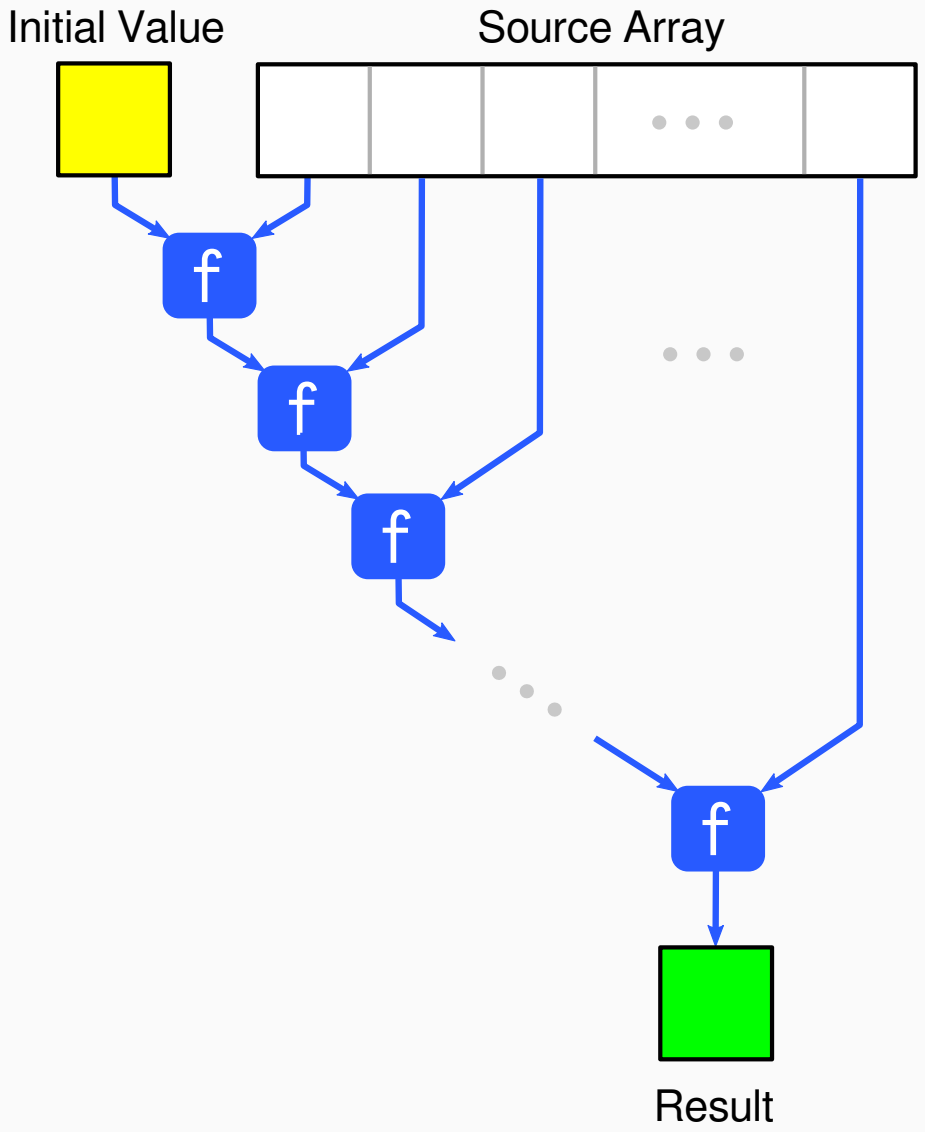
reduce executes a reducer function on each member of the array resulting on a single value.

```
[1, 2, 3, 4].reduce((sum, n) => sum + n, 0)   // -> 10
[1, 2, 3, 4].reduce((mult, n) => mult * n, 1) // -> 24

const myjoin = (str_array, sep) =>
  str_array.slice(1)
    .reduce((accum, s) => accum + sep + s, str_array[0])
myjoin(["a", "b", "c"], ";")  // -> "a;b;c"
```

reduceRight works exactly like reduce, but starts from the last element and works backwards.

reduce diagram



The **reduce** Algorithm

The **reduce** algorithm could be implemented as follows:

```
function reduce(array, fn, initialValue) {
  let accum = initialValue
  for (let i = 0; i < array.length; i++) {
    accum = fn(accum, array[i], i, array)
  }
  return accum
}
```

some and **every**

```
<Array>.some((item, index, array) => {
  // return true if item satisfies condition
})
<Array>.every((item, index, array) => {
  // return true if item satisfies condition
})
```

The method **some** tests whether at least one element in the array satisfies the condition implemented by the provided function. The method **every** tests whether all elements of the array satisfy the condition.

```
const allStrings   = arr => arr.every(s => typeof s === "string")
const oneNegative  = num => nums.some(n => n < 0)
const allUpperCase = str =>
  [...str].every(c => c === c.toUpperCase())
const nullProps = obj =>
  Object.entries(obj).every(prop => prop[1] !== null)
```

some and **every**

The algorithms for **some** and **every** could be implemented as follows:

```
function some(array, func) {
  for (let i = 0; i < array.length; i++) {
    if (func(array[i], i, array)) return true
  }
  return false
}

function every(array, func) {
  for (let i = 0; i < array.length; i++) {
    if (!func(array[i], i, array)) return false
  }
  return true
}
```

Table of Contents

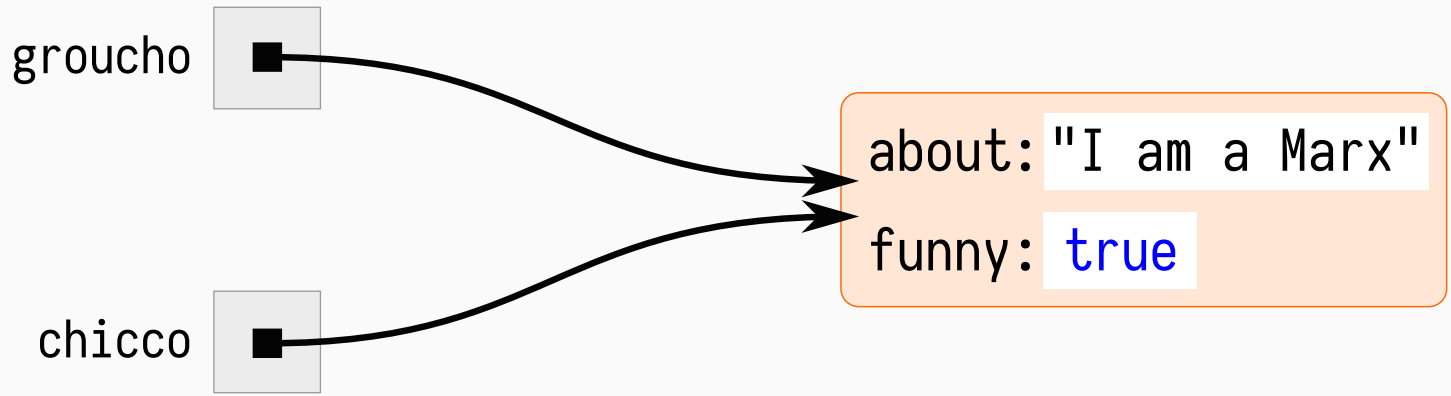
Copying Objects

Copying Values

With the exception of basic types (**Number**, **Boolean**, and **String**), variables hold references to objects (i.e. pointers to memory locations).

Assigning a variable to another *just copies the reference* (not the content).

```
let groucho = { about: "I am a Marx", funny: true };
let chicco = groucho;
```



Copying Objects

Two variables can point to the same object, so using either variable affects the same memory region:

```
let flower1 = { kind: "rose", color: "red" };
let flower2 = flower1;
flower2.color = "yellow";
console.log(flower1.color);  // -> yellow
```

One way to copy an object would be to explicitly copy its properties:

```
let flower3 = {};
for (let prop in flower1) {
  flower3[prop] = flower1[prop];
}
```

(But what if the properties have properties themselves?)

Merging objects

A special function does this kind of copy:

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

It merges all properties of objects `src1`, `src2`, etc. into the object `dest`.

If `dest` has the same properties as `srcN`, they are overwritten.

```
let model = { brand: "Tesla", name: "X" };
let configuration = { color: "red", battery: "90kWh" };

let mycar = Object.assign({}, model, configuration);
```

Copying with Spread

When can ”spread” an existing object’s properties when creating a new one:

```
var obj1 = { foo: "bar", x: 42 };
var obj2 = { foo: "bazzz", y: 13 };
var clone1 = { ...obj1 };  // { foo: "bar", x: 42 }
```

We can also merge, the properties of two objects:

```
var clone2 = { ...obj1, ...obj2 };  // { foo: "bazzz", x: 42, y: 13 }
```

And we can change some of the properties of the original object:

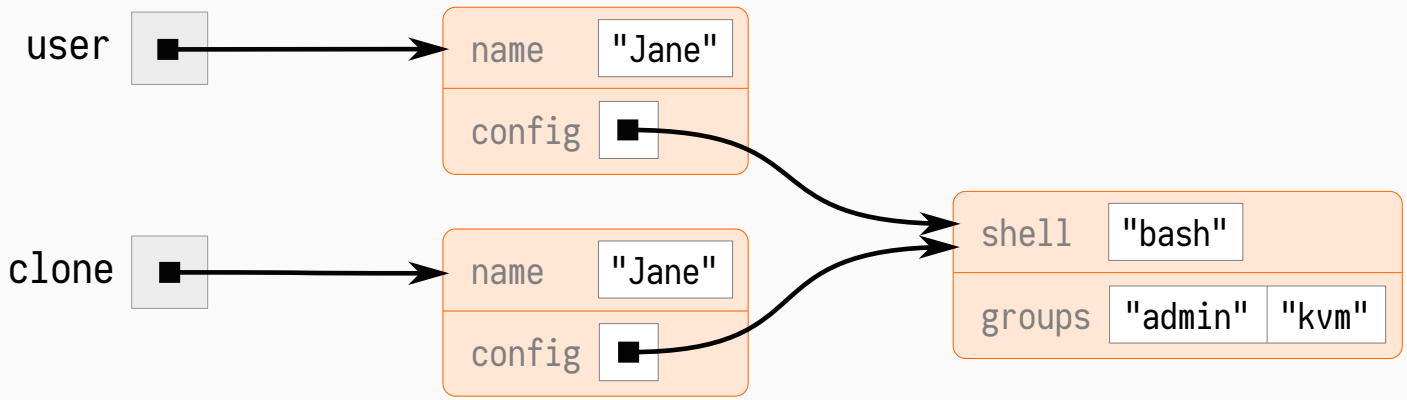
```
var clone3 = { ...obj2, y: 17 };  // { foo: "bazzz", y: 17 }
```

The spread operator always does a shallow copy (first level copy)

Shallow Copy

What if we clone an object with properties referencing other objects?

```
let user = {
    name: "Jane",
    config: {
        shell: "bash",
        groups: ["admin", "kvm"]
    }
}
let clone = Object.assign({}, user)
```



Deep Copy

Deep copy: not only duplicate the initial object, but also clone properties recursively.

The deep copy algorithm is tricky since there are many properties of an object that have special meaning and shouldn't be cloned (i.e. `prototype`).

It is better to use an existing library to do deep copies:

Using the `lodash` library:

```
const _ = require('lodash');
var objects = [{ 'a': 1 }, { 'b': 2 }];
var deep = _.cloneDeep(objects);
console.log(deep[0] === objects[0]);  // -> false
```