JavaScript 1 - Module 2

Strings and Logic

Arrays

Objects

Functions

Solutions for JS1 Lesson 2.2 Arrays

Exercise 1

Make an array with the first 10 Fibonacci numbers: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34

a) What is the value of the seventh Fibonacci number? Console log out the answer.

b) Make a variable `sum`. Use a `for` loop to go over the array, and add up the value of the numbers. Console log out the sum.

```javascript
// Exercise 1
console.log("Exercise 1");

const fib = [0, 1, 1, 2, 3, 5, 8, 13, 21, 34];
// See lesson 1.4 for how to calculate this...

// a
console.log(fib[6]); // 8, remember array indexes start at 0
// However: Normally we refer to fibonaccinumber as Fn
// Making F0 = 0, F1 = 1, F2 = 1, ..., F6 = 8, F7 = 13, F8, = 21, ...,
// ie. starting on 0, like an array, So F7 is not the same as the 7th number...
console.log(fib[7]); // 13

// b
let sum = 0;

for (let i = 0; i < fib.length; i++) {
    sum += fib[i];
}
console.log(sum); // 88
```

Exercise 2

Given the Array:

```
let myNumbers = [13, -2, 18, 4, 42, 12, 9, -21, -3];
```

a) Console log out how many items the Array holds.

b) Remove the last item in the Array.

c) Add the number `14` to the end of the Array.

d) Add the number `3` to the start of the Array.

e) Use a for loop to go over the Array and find the largest number.

> Tip: Use a variable largest (or max) and compare every item to the largest, update largest if the item is larger.

```javascript
// Exercise 2
console.log("Exercise 2");

let myNumbers = [13, -2, 18, 4, 42, 12, 9, -21, -3];

// a
console.log(myNumbers.length); // 9

// b
myNumbers.pop();

// c
myNumbers.push(14);

// d
myNumbers.unshift(3);
console.log(myNumbers); // Just checking... :-)
// Array(10) [ 3, 13, -2, 18, 4, 42, 12, 9, -21, 14 ]

// e - almost level 2 ;-)
let max = -Infinity; // Nothing is less than negative infinity
for (let num of myNumbers) {
    if (num > max) {
        max = num;
    }
}
console.log("The largest number is " + max); // The largest number is 42
```

Exercise 3

Given the Array:

```
let vegetables = ['Cabbage', 'Turnip', 'Radish', 'Carrot'];
```

a) Console log out how many items the Array holds.

b) Add `"Tomato"` to the end of the Array.

c) Add `"Cucumber"` to the start of the Array.

d) Use a `for` loop to go over the Array, and for each item, console log out the value of the item and how many letters it contains (eg. "Tomato contains 6 letters").

```
// Exercise 3
console.log("Exercise 3");

let vegetables = ['Cabbage', 'Turnip', 'Radish', 'Carrot'];

// a
console.log(vegetables.length); // 4

// b
vegetables.push("Tomato");

// c
vegetables.unshift("Cucumber");
console.log(vegetables); // Just to check...
// Array(6) [ "Cucumber", "Cabbage", "Turnip", "Radish", "Carrot", "Tomato" ]

// d
for (let item of vegetables) {
    console.log(`${item} contains ${item.length} letters`);
}

// d, equally correct
for (let i = 0; i < vegetables.length; i++) {
    let output = vegetables[i] + " contains " + vegetables[i].length + " letters";
    console.log(output);
}
```

Exercise 4

Given the array:

```
let mixed = ["Dill", 42, true, , 13, "13", null];
```

a) Make a `for` loop to loop over all items and, for each item console log out the index, value and data type of the element.

b) Add the string `"Hello, World!"` to the `undefined` element in the Array `mixed`, then console.log out `mixed`.

c) Make another `for` loop and add up all the items in the Array that has a numeric data type. Console log the sum.

```
// Exercise 4
console.log("Exercise 4");

// a
let mixed = ["Dill", 42, true, , 13, "13", null];
for (let i = 0; i < mixed.length; i++) {
    console.log(i + " contains " + mixed[i] + " with the datatype " + typeof mixed[i]);
}

// b
mixed[3] = "Hello, World!";
console.log(mixed);
// Array(7) [ "Dill", 42, true, "Hello, World!", 13, "13", null ]

// c
let numericSum = 0;
for (let item of mixed) {
    if (typeof item === 'number') {
        numericSum += item;
    }
}
console.log(numericSum); // 55
```

Exercise 5

Given the two Arrays:

```
let vegetables = ['Cabbage', 'Turnip', 'Radish', 'Carrot'];
let fruits = ["Banana", "Orange", "Lemon", "Apple", "Mango"];
```

a) Merge the 2 Arrays into a new Array, `greeneries`.

b) Sort the merged Array then console log it.

> Tip: Look at the `Array.sort()` method.

c) Console log out the first element in the merged, then sorted Array.

d) Console log out the last element in the merged, then sorted Array.

e) Sort the merged Array in reverse order.

> Tip: Look at the `Array.reverse()`, and remember to `sort()` it first.

```
// Exercise 5
console.log("Exercise 5");

vegetables = ['Cabbage', 'Turnip', 'Radish', 'Carrot'];
// Tip: Do not redeclare if you used let vegetables = [...] in #3 above
let fruits = ["Banana", "Orange", "Lemon", "Apple", "Mango"];

// a
let greeneries = vegetables.concat(fruits);

// b
greeneries.sort();
console.log(greeneries);
// Array(9) [ "Apple", "Banana", "Cabbage", "Carrot", "Lemon", "Mango", "Orange", "Radish", "Turnip" ]

// c
console.log(greeneries[0]); // Apple

// d
console.log(greeneries[greeneries.length - 1]); // Turnip

// e
// Since greeneries is already sorted above:
greeneries.reverse();
console.log(greeneries);
// Array(9) [ "Turnip", "Radish", "Orange", "Mango", "Lemon", "Carrot", "Cabbage", "Banana", "Apple" ]
```

Exercise 6 - Level 2

Given the Array of numbers from Exercise 2

```
let myNumbers = [13, -2, 18, 4, 42, 12, 9, -21, -3];
```

a) Sort `myNumbers` using the `Array.sort()` method from above. Console log the sorted Array. Did you get the result you expected?

b) Use a *compare function* to sort the numbers by numeric value, rather then alphabetically.

c) Sort the numbers in reverse order, ie. largest first, by adjusting the compare function.

```
// Exercise 6 - Level 2
console.log("Exercise 6 - Level 2");

myNumbers = [13, -2, 18, 4, 42, 12, 9, -21, -3];

// a
myNumbers.sort();
console.log(myNumbers);
// [ -2, -21, -3, 12, 13, 18, 4, 42, 9 ] > The array is sorted alphabetically

// b
myNumbers.sort(function(a, b){return a - b});
console.log(myNumbers);
// [ -21, -3, -2, 4, 9, 12, 13, 18, 42 ] > correct, ascending order

// c
myNumbers.sort(function(a, b){return b - a});
console.log(myNumbers);
// [ 42, 18, 13, 12, 9, 4, -2, -3, -21 ] > correct, descending order
```

Bonus:

Use `pop` (or `shift`) to go through an `array` using a `while` loop:

HTML:

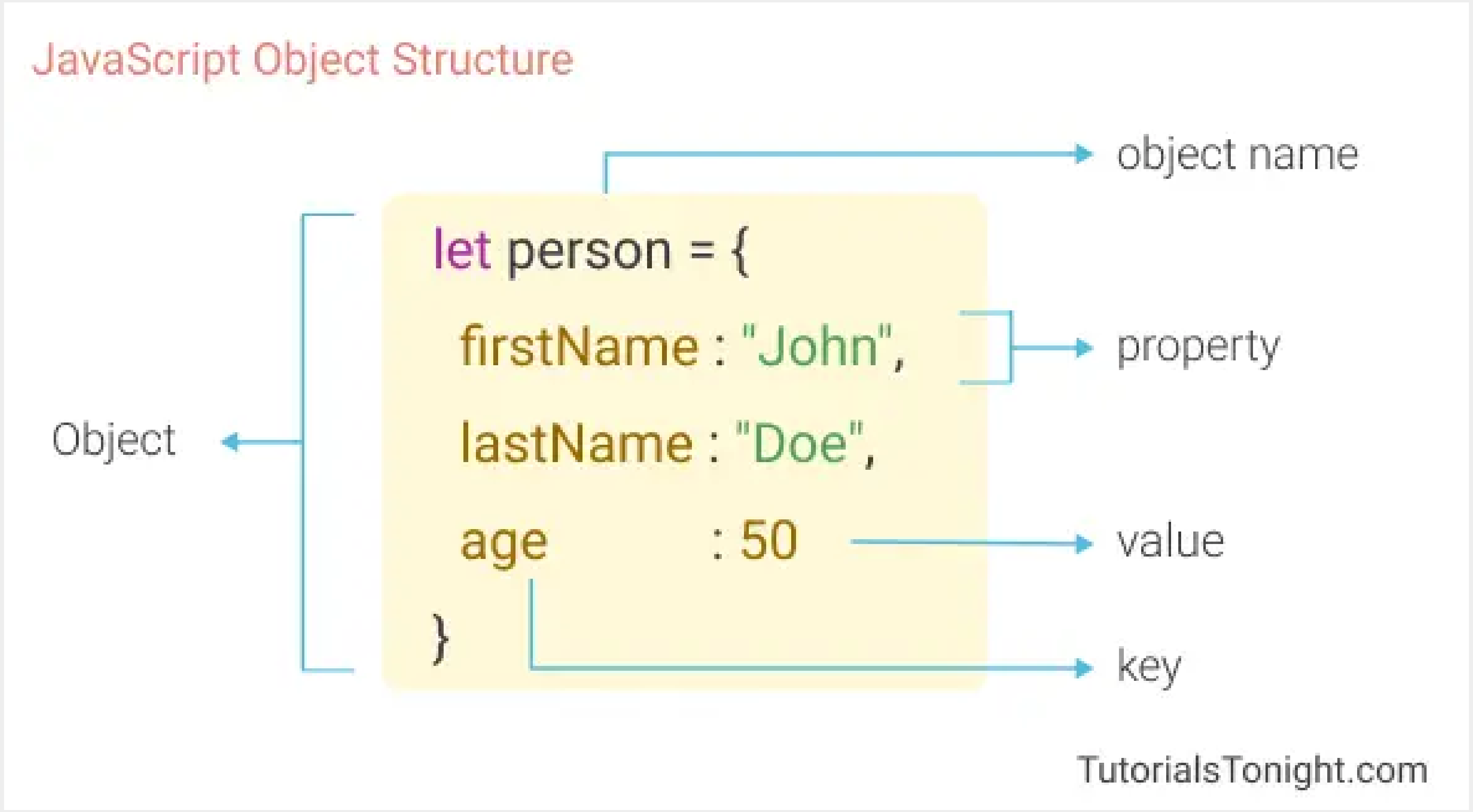
```html
<h1>Vegetables</h1>
<ul id="output"></ul>
```

JS:

```js
const out = document.querySelector("ul#output");
const data = ["Carrots", "Broccoli", "Zucchini", "Cabbage", "Spinach",
"Tomatoes", "Cucumber", "Lettuce", "Mushrooms", "Green Beans"];
data.sort(); // Don't need a compare function since we use uniform strings
data.reverse(); // Since we pop items (ie. start at the back)
//console.log(data);
while (item = data.pop()){
    out.innerHTML += `<li>${item}</li>`;
}
```

Codepen

`null`, objects and arrays of objects



```
let person = {
  firstName : "John",
  lastName : "Doe",
  age        : 50
}
```

Source

Null

In JavaScript null is "nothing". It is supposed to be something that doesn't exist.

Unfortunately, in JavaScript, the data type of null is an object, *it should have been null*.

```
let number = 42;
console.log (number, typeof number);
// Expected output: 42 number

number = null;
console.log (number, typeof number);
// Expected output: null object

number = undefined;
console.log (number, typeof number);
// Expected output: undefined undefined
```

Null, cont.

The value null is written with a literal: `null`.

`null` is not an identifier for a property of the global object, like `undefined` can be.

Instead, `null` expresses a lack of identification, indicating that a variable points to no object.

In APIs, `null` is often retrieved in a place where an object can be expected but no object is relevant.

Difference between null and undefined

When checking for `null` or `undefined`, beware of the differences between equality (`==`) and identity (`===`) operators, as the former performs type-conversion.

```
typeof null           // "object" (not "null" for legacy reasons)
typeof undefined      // "undefined"
null === undefined    // false
null == undefined    // true
null === null         // true
null == null          // true
!null                 // true
isNaN(1 + null)       // false, ie. (1 + null) equals 1
isNaN(1 + undefined) // true, ie. (1 + undefined) equals NaN
```

JavaScript Objects

JavaScript is designed on a simple object-based paradigm.

An object is a collection of properties, and a property is an association between a name (or key) and a value.

A property's value can be a `function`, in which case the property is known as a `method`.

In addition to objects that are predefined in the browser, you can define your own objects.

Real Life Objects, Properties, and Methods

Objects in JavaScript, just as in many other programming languages, can be compared to objects in real life. The concept of objects in JavaScript can be understood with real life, tangible objects.

*In real life, a **car** is an object.*

All cars have the same properties, but the *property values* differ from car to car.

All cars have the same methods, but the methods are performed at different times.

Car as an object

A car has properties like weight and color, and methods like start and stop:

Properties	Methods
car.name="Fiat"	car.start()
car.model="500"	car.drive()
car.weight="850kg"	car.brake()
car.color="white"	car.stop()

JavaScript Objects

You have already learned that JavaScript variables are containers for data values. This code assigns a simple value (Fiat) to a variable named car:

```
var car = "Fiat";
```

Objects are variables too. But objects can contain many values. This code assigns many values (Fiat, 500, white) to a variable named car:

```
var car = {type:"Fiat", model:"500", color:"white"};
```

The values are written as `name` : `value` pairs (name and value separated by a colon). JavaScript objects are containers for named values called properties or methods.

Object Definition

You define (and create) a JavaScript object with an object literal:

```
var person = {firstName:"John", lastName:"Doe", age:42, eyeColor:"blue"};
```

Spaces and line breaks are not important. An object definition can span multiple lines, which is often better for readability:

```
var person = {
  firstName: "John",
  lastName: "Doe",
  age: 42,
  eyeColor: "blue"
};
```

Object Properties

The name:values pairs in JavaScript objects are called properties:

Property	Property Value
firstName	"John"
lastName	"Doe"
age	42
eyeColor	"blue"

Accessing Object Properties

You can access object properties in two ways:

`objectName.propertyName` (called **dot-notation**)

or

`objectName["propertyName"]` (called **bracket notation**)

```
var person = {
    firstName:"John",
    lastName:"Doe",
    age:50,
    eyeColor:"blue"
};
console.log (person.firstName);   // "John"
console.log (person["lastName"]); // "Doe"
```

Object Methods

Objects can also have methods.

Methods are actions that can be performed on objects.

Methods are stored in properties as `function` definitions:

```
var person = {
  firstName: "John",
  lastName : "Doe",
  id       : 5566,
  fullName : function() {
    return this.firstName + " " + this.lastName;
  }
};
```

> More on functions in Lesson 2.4

Accessing Object Methods

You access an object method with the following syntax: `objectName.methodName()`

Given the person variable on the last slide:

```
let name = person.fullName();

console.log (name); // "John Doe"
```

If you access a method without the () parentheses, it will return the function definition:

```
let name2 = person.fullName;

console.log(name2);
// Expected output:
// "function() {\n    return this.firstName + \" \" + this.lastName;\n  }"
```

Sidenote: The `this` Keyword

In a function definition, `this` refers to the "owner" of the function.
(Or said another way: The object from which the method has been called.)

In the example above, `this` is the `person` object that "owns" the fullName function.

In other words, `this.firstName` means the `firstName` property of *this* object.

Arrays of objects

Given a situation where we want to make many objects of a certain kind, and loop through those, we may want to put them in an array.

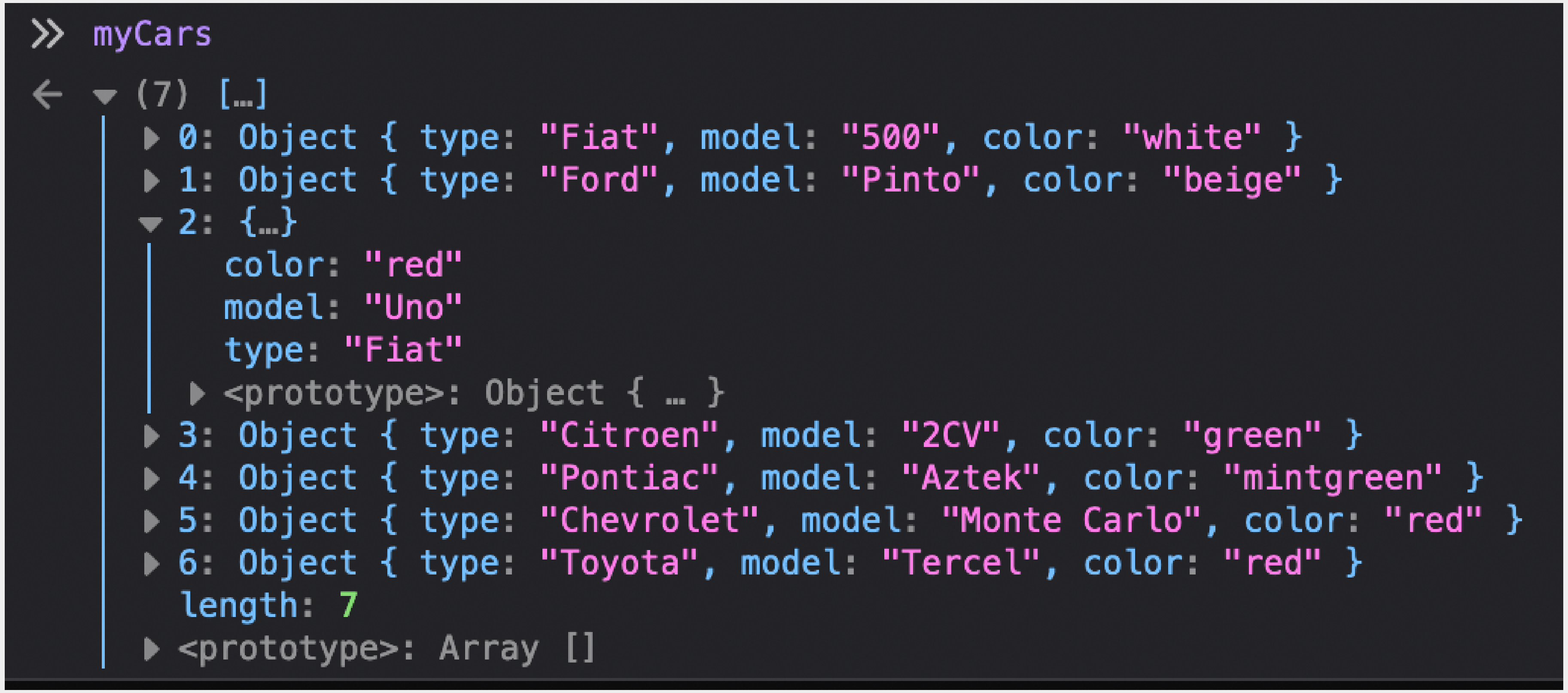
So, given:

```
let myFiat500 =  { type:"Fiat", model:"500", color:"white" };
let myFord =  { type:"Ford", model:"Pinto", color:"beige" };
let myFiatUno =  { type:"Fiat", model:"Uno", color:"red" };
let myCitroen =  { type:"Citroen", model:"2CV", color:"green" };
let anotherCar =  { type:"Pontiac", model:"Aztek", color:"mintgreen" };
let andAnother =  { type:"Chevrolet", model:"Monte Carlo", color:"red" };
let replacement =  { type:"Toyota", model:"Tercel", color:"red" };
```

We can now put those cars into an array:

```
let myCars = [myFiat500, myFord, myFiatUno,
myCitroen, anotherCar, andAnother, replacement];
```

If you use console to log those, you get an Array with all 7 car Objects in is, and you may expand to see the properties of each Object:



```
>> myCars
← ▼ (7) […]
    ▶ 0: Object { type: "Fiat", model: "500", color: "white" }
    ▶ 1: Object { type: "Ford", model: "Pinto", color: "beige" }
    ▼ 2: {…}
        color: "red"
        model: "Uno"
        type: "Fiat"
      ▶ <prototype>: Object { … }
    ▶ 3: Object { type: "Citroen", model: "2CV", color: "green" }
    ▶ 4: Object { type: "Pontiac", model: "Aztek", color: "mintgreen" }
    ▶ 5: Object { type: "Chevrolet", model: "Monte Carlo", color: "red" }
    ▶ 6: Object { type: "Toyota", model: "Tercel", color: "red" }
      length: 7
    ▶ <prototype>: Array []
```

Arrays of objects, cont.

To access the data you can use:

```
console.log (myCars[2]);
// Object { type: "Fiat", model: "Uno", color: "red" }

console.log (myCars[2].type);  // "Fiat"
console.log (myCars[2].model); // "Uno"
console.log (myCars[2].color); // "red"
console.log (myCars[2].year);  // undefined
```

> Note that since the car Object doesn't have a property `year`, `myCars[2].year` returns `undefined` (and not `null`).

Adding properties:

If you need to change a property:

```
myCars[2].color = "blue"; // Paintjob

console.log (myCars[2]);
// Object { type: "Fiat", model: "Uno", color: "blue" }
```

You may add an extra property to an Object:

```
myCars[2].year = 1993;

console.log (myCars[2]);
// Object { type: "Fiat", model: "Uno", color: "blue", year: 1993 }

console.log (myCars[3]);
// Object { type: "Citroen", model: "2CV", color: "green" }
```

> Note that the other cars did not get the extra property.

Listing all properties of an object

One way of listing all preperties in an object is by using the `for...in` loop:

Using one of the cars from above, `anotherCar` :

```
let anotherCar =  { type:"Pontiac", model:"Aztek", color:"mintgreen" };

for (let prop in anotherCar) { // looping over all properties in anotherCar
    console.log (prop + " : " + anotherCar[prop]);  // key : value
}

// Expected output:
// type : Pontiac
// model : Aztek
// color : mintgreen
```

Listing all items in an Array

One way of listing all preperties in an object is by using the `for...of` loop:

Using the `myCars` with all cars from above:

```
for (let car of myCars) { // looping over all items of myCars
    console.log ( car );  // listing each item
}
// Expected output:
// Object { type: "Fiat", model: "500", color: "white" }
// Object { type: "Ford", model: "Pinto", color: "beige" }
// Object { type: "Fiat", model: "Uno", color: "blue", year: 1993 }
// Object { type: "Citroen", model: "2CV", color: "green" }
// Object { type: "Pontiac", model: "Aztek", color: "mintgreen" }
// Object { type: "Chevrolet", model: "Monte Carlo", color: "red" }
// Object { type: "Toyota", model: "Tercel", color: "red" }
```

Listing all properties of all objects in an Array

Again, using the `myCars` with all cars from above, now in a double loop:

```
let output = "";
for (let car of myCars) { // looping over all items of myCars
    for (let prop in car) { // then looping over all properties in each car
        output += car[prop] + " "; // adding property value to the output
    }
    output += "\n"; // Adding line break after listing all props for a car
}
console.log(output); // Expected output:
// Fiat 500 white
// Ford Pinto beige
// Fiat Uno blue 1993
// Citroen 2CV green
// Pontiac Aztek mintgreen
// Chevrolet Monte Carlo red
// Toyota Tercel red
```

JavaScript JSON

JSON is a format for storing and transporting data.

JSON is often used when data is sent from a server to a web page.

What is JSON?

- JSON stands for **J**ava**S**cript **O**bject **N**otation
- JSON is a lightweight data interchange format
- JSON is language independent **
- JSON is "self-describing" and easy to understand

** The JSON syntax is derived from JavaScript object notation syntax, but the JSON format is text only. Code for reading and generating JSON data can be written in any programming language.

JSON Example

This JSON syntax defines an employees object: an array of 3 employee records (objects):

```
{
    "employees":[
        {"firstName":"John", "lastName":"Doe"},
        {"firstName":"Anna", "lastName":"Smith"},
        {"firstName":"Peter", "lastName":"Jones"}
    ]
}
```

The JSON Format Evaluates to JavaScript Objects

The JSON format is syntactically identical to the code for creating JavaScript objects.

Because of this similarity, a JavaScript program can easily convert JSON data into native JavaScript objects.

JSON Syntax Rules

- Data is in name/value pairs
- Data is separated by commas
- Curly braces hold objects
- Square brackets hold arrays

JSON Data - A Name and a Value

JSON data is written as name/value pairs, just like JavaScript object properties.

A name/value pair consists of a field name (in double quotes), followed by a colon, followed by a value:

```
"firstName":"John"
```

> NOTE: JSON names **require** double quotes. JavaScript names do not.

JSON Objects

JSON objects are written inside curly braces, `{ }`.

Just like in JavaScript, objects can contain multiple name/value pairs:

```
{"firstName":"John", "lastName":"Doe"}
```

JSON Arrays

JSON arrays are written inside square brackets, `[ ]`.

Just like in JavaScript, an array can contain objects:

```
"employees":[
  {"firstName":"John", "lastName":"Doe"},
  {"firstName":"Anna", "lastName":"Smith"},
  {"firstName":"Peter", "lastName":"Jones"}
]
```

In the example above, the object `employees` is an array. It contains three objects.

Each object is a record of a person (with a first name and a last name).

Converting a JSON Text to a JavaScript Object

A common use of JSON is to read data from a web server, and display the data in a web page. For simplicity, this can be demonstrated using a string as input.

First, create a JavaScript string containing JSON syntax:

```
var text = '{ "employees" : [' +
'{ "firstName":"John" , "lastName":"Doe" },' +
'{ "firstName":"Anna" , "lastName":"Smith" },' +
'{ "firstName":"Peter" , "lastName":"Jones" } ]}';
```

Then, use the JavaScript built-in function `JSON.parse()` to convert the string into a JavaScript object:

```
var obj = JSON.parse(text);

console.log(obj.employees[1].lastName); // "Smith"
```

Sources and resources

JavaScript Objects

JavaScript Guide > Working with objects

JavaScript JSON (JavaScript Object Notation)

JavaScript reference > Standard built-in objects > null

Todos

Github

JS1 Lesson 2.3 Objects

Mollify

Read null, Objects and arrays of objects, and do the Lesson Task.