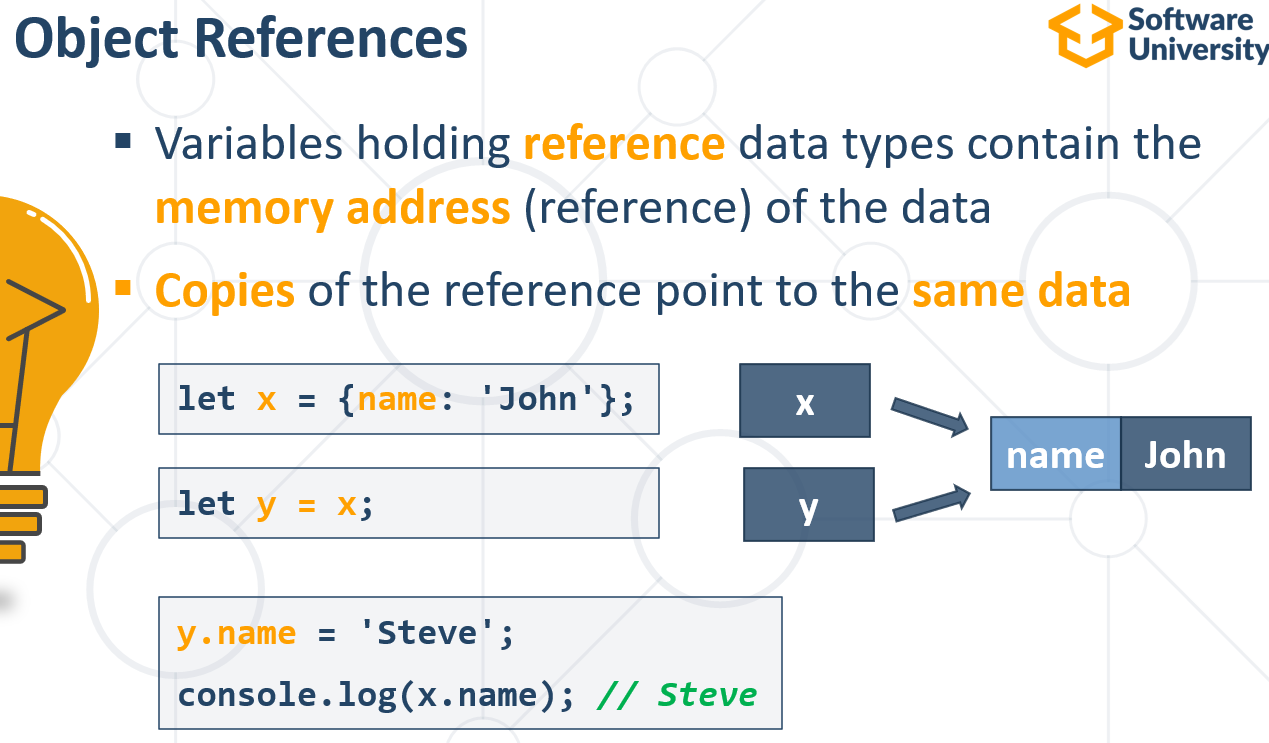
**Lab: Objects & Composition**

Problems for in-class lab for the ["JavaScript Advanced" course @ SoftUni](https://softuni.bg/trainings/3217/js-advanced-january-2021). Submit your solutions in the SoftUni judge system at <https://judge.softuni.bg/Contests/2758/Objects-and-Composition-Lab>.

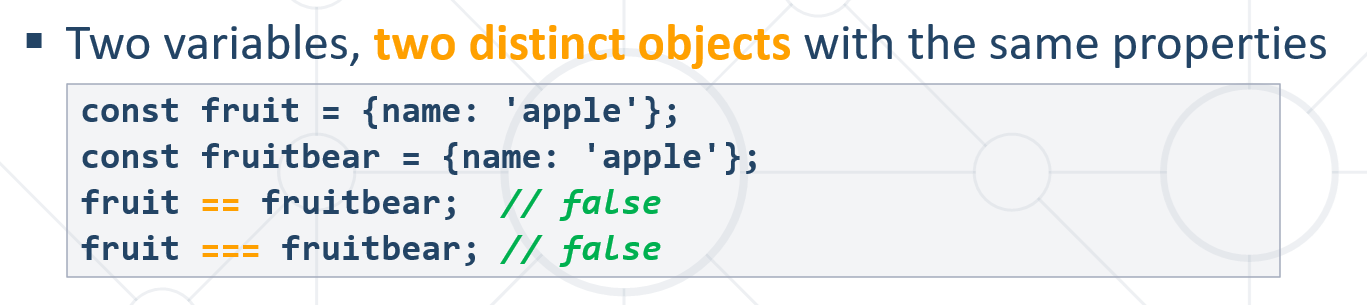
# I. Теория

## Objects in JavaScript

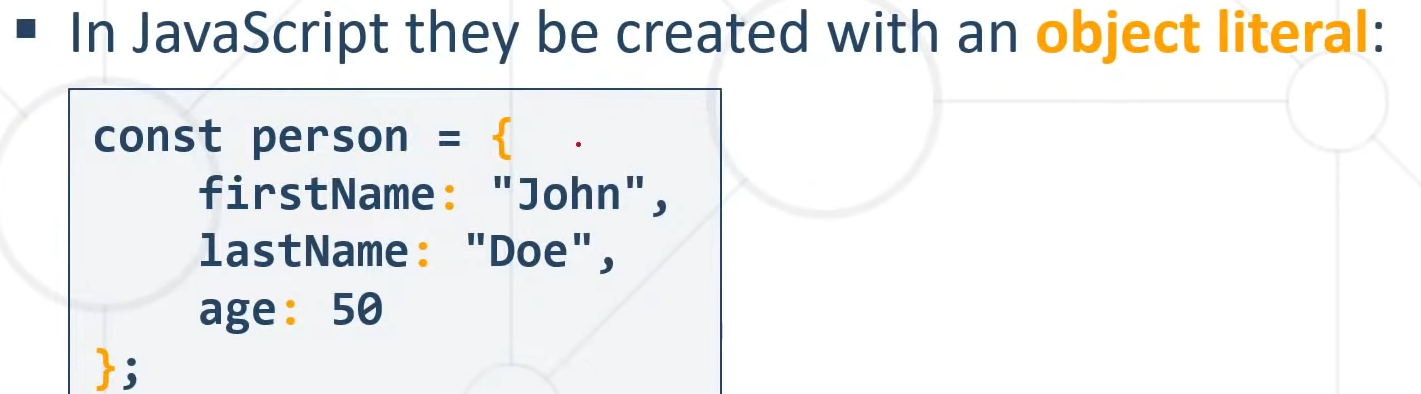
Обектите в JS могат да се разглеждат като съвкупност от полета (collection of fields) и функции, които се наричат свойства („properties“). Под “property” може да се разбира асоциацията с ключ и стойност (key and value), които имаме при Map-овете в Jаvа. Както в Java, така и в JS, стойностите са асоциирани с някакъв ключ. За разлика от Java, в JS асоциативните масиви не са колекция от „еднотипни“ properties (например само String), а са група от properties, които се отнасят за дадено явление **(с други думи отнасят се за обект). Обектите са референтен тип данни, т.е. пазят се в Heap-a. В Stack-a се пази само референция към адреса на обекта в heap-a.**



Обектите се сравняват по рефенреция (както масивите), което означава, че два обекта, които съдържат едни и същи свойства, на практика са различни неща :



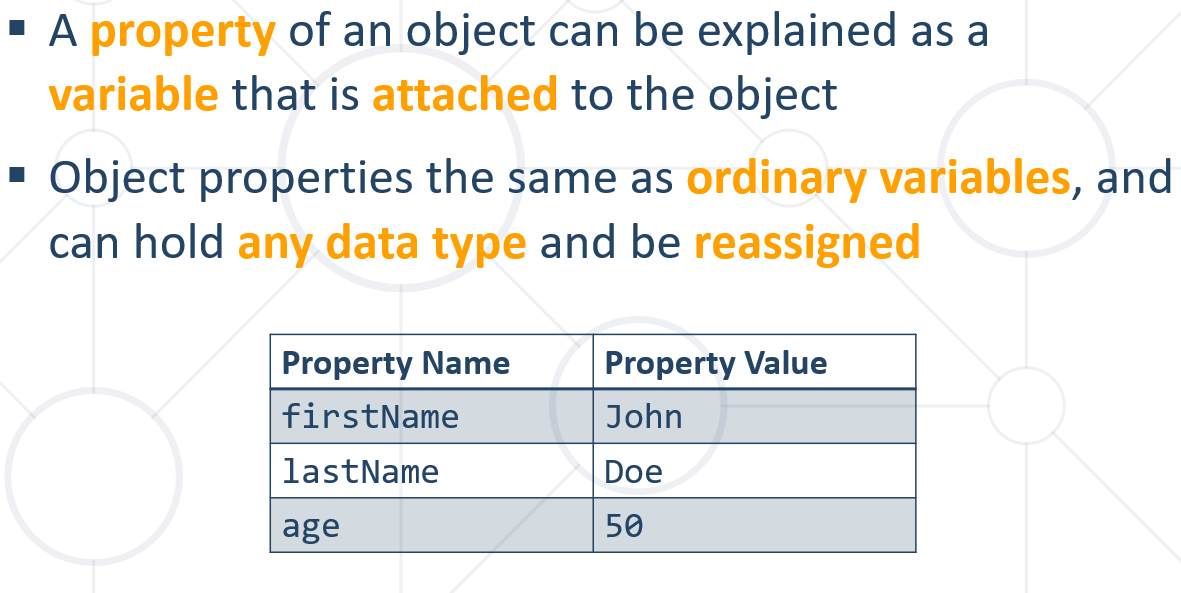
Инициализиране на обект в JS става с полета (свойства или properties), които се състоят от идентификатор (firstName) и стойност – литерала String (“John“) например.



В този случай създаваме обекта Person.

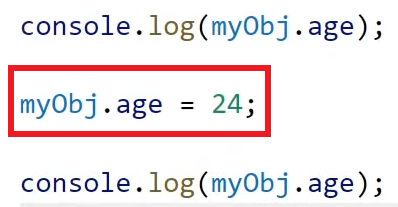
### Object properties

Свойствата на един обект могат да се разглеждат като променливи, които са прикачени за него. Тези променливи могат да съдържат различен тип данни и да бъдат ре-инициализирани.



### Аccessing and re-assigning objects properties

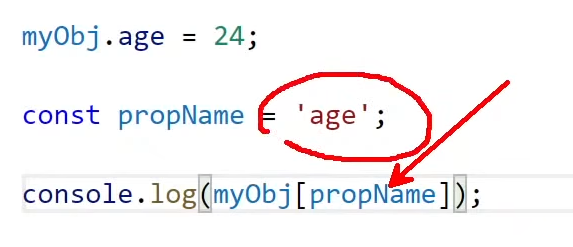




Друг начин да достъпим стойност е по неговия ключ. **Всички ключове в JS са от тип “String”**, поради което следната операция е възможна :



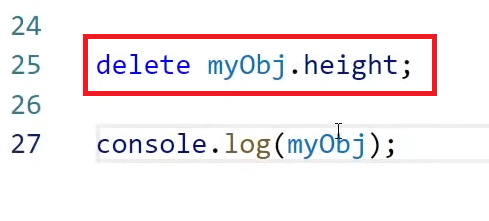
**[] е индексиращ оператор, който обикновено използваме тогава, когато не знаем как точно се казва нашето свойство**, което искаме да достъпим (ако например то идва от конзолата, и сме го запаметили в някаква променлива). Тогава ще го достъпим чрез [] :



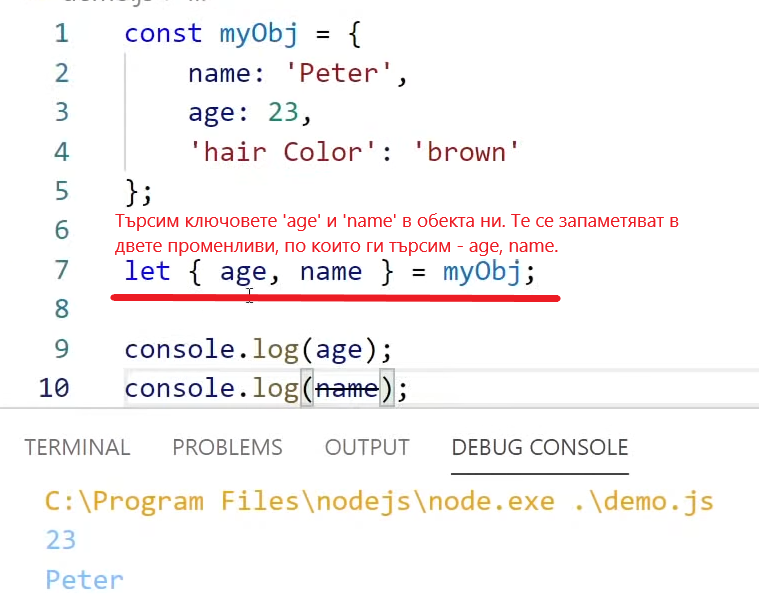
Добавянето на property става по същия начин като достъпването. **Ако не съществува такова свойство** до момента в обекта, то ще се създаде автоматично. Например:



### Deleting objects property



### Деструктириране на обект по ключове

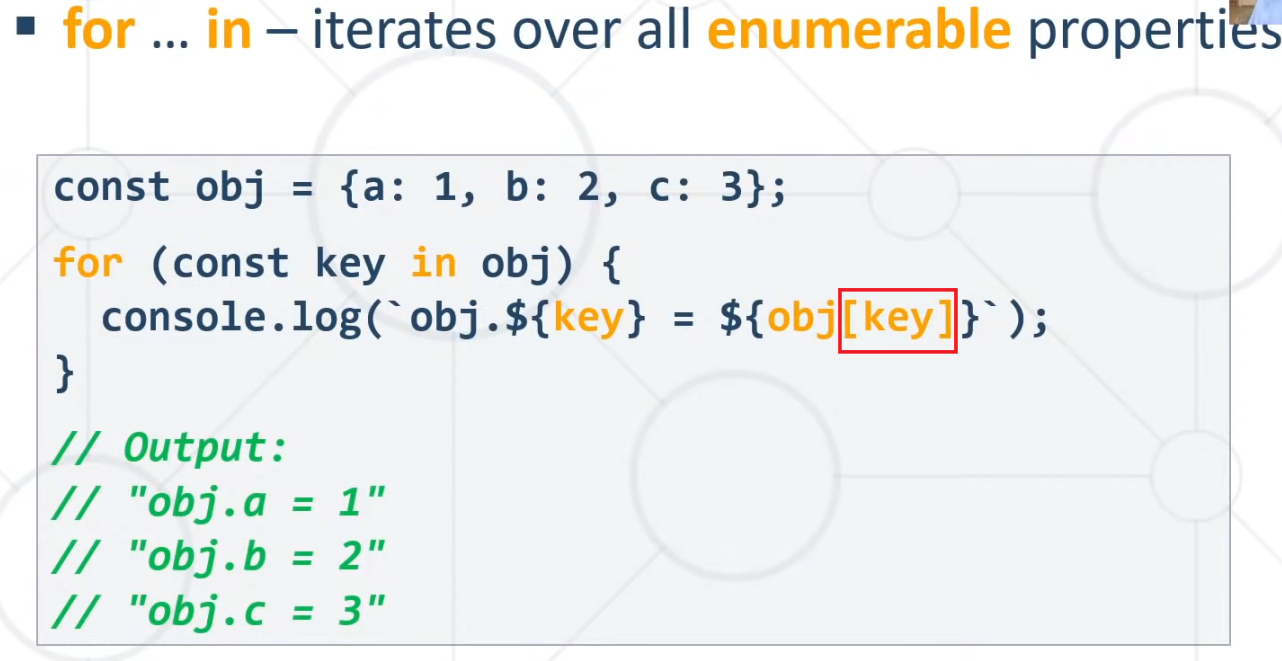


Ако искаме да променим ключа на „property-то“ в новата променлива, правим следното :



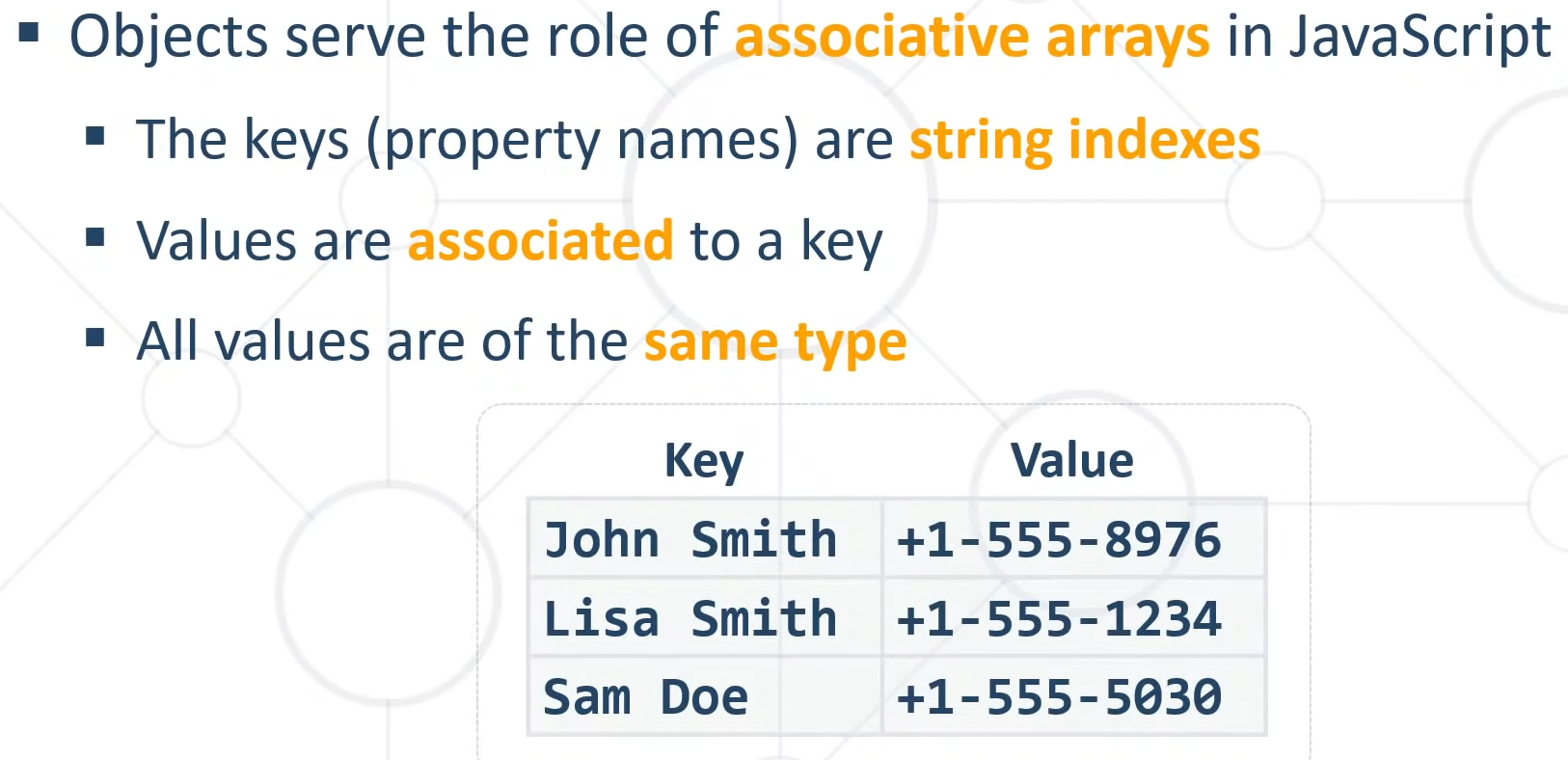
### For… in - обхождане на обект

**Обхождането на обект става по неговите ключове.** Тъй като **не знаем каква стойност** **стои като ключ**, трябва да използваме **индексиращия оператор [],** който ще ни върне value-то срещу всеки ключ.



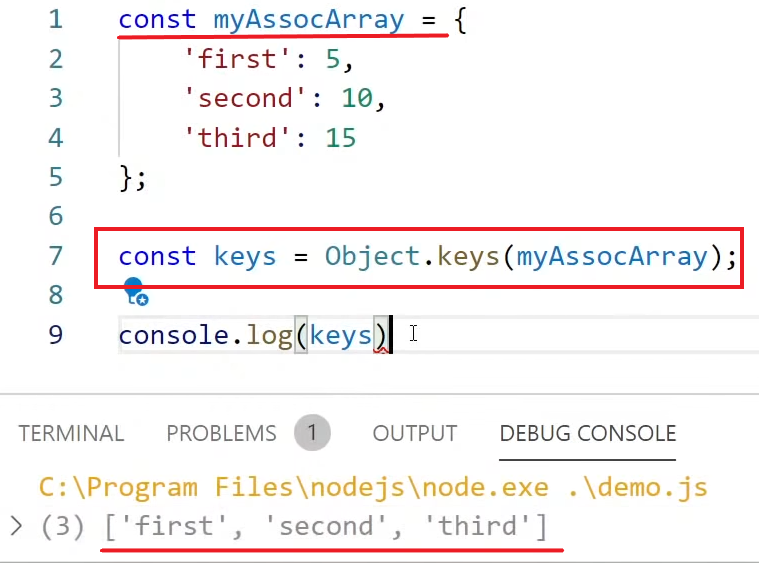
## Objects as Associative Arrays

Обектите могат да се предсатвят и като асоциативни масиви (още map и hashTable). Те се достъпват по абсолютно същия начин както достъпваме един обект, но **разликата е в приложението им.** Използвайки обект обикновено **всички стойности**, които той съдържа, сочат към едно „явление“ и **са от различен тип**, докато ако използваме обекта като map, неговите **стойности е прието да са от един и същи тип.**

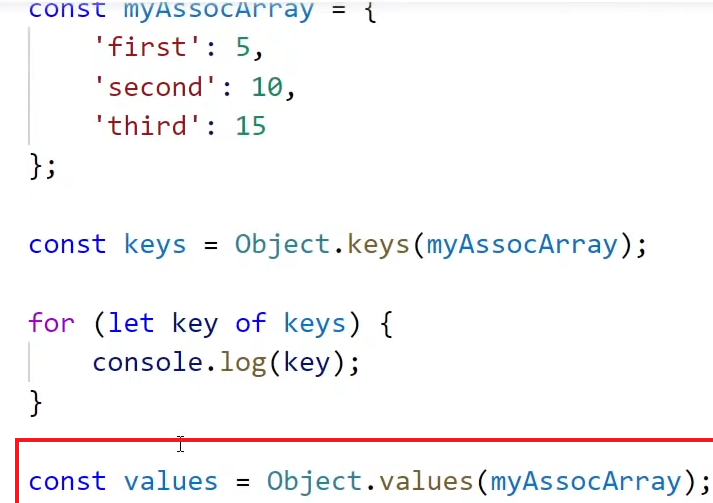


### Вградени “object” методи, които можем да използваме

#### Изкарване на keys и values в масив/итератор :

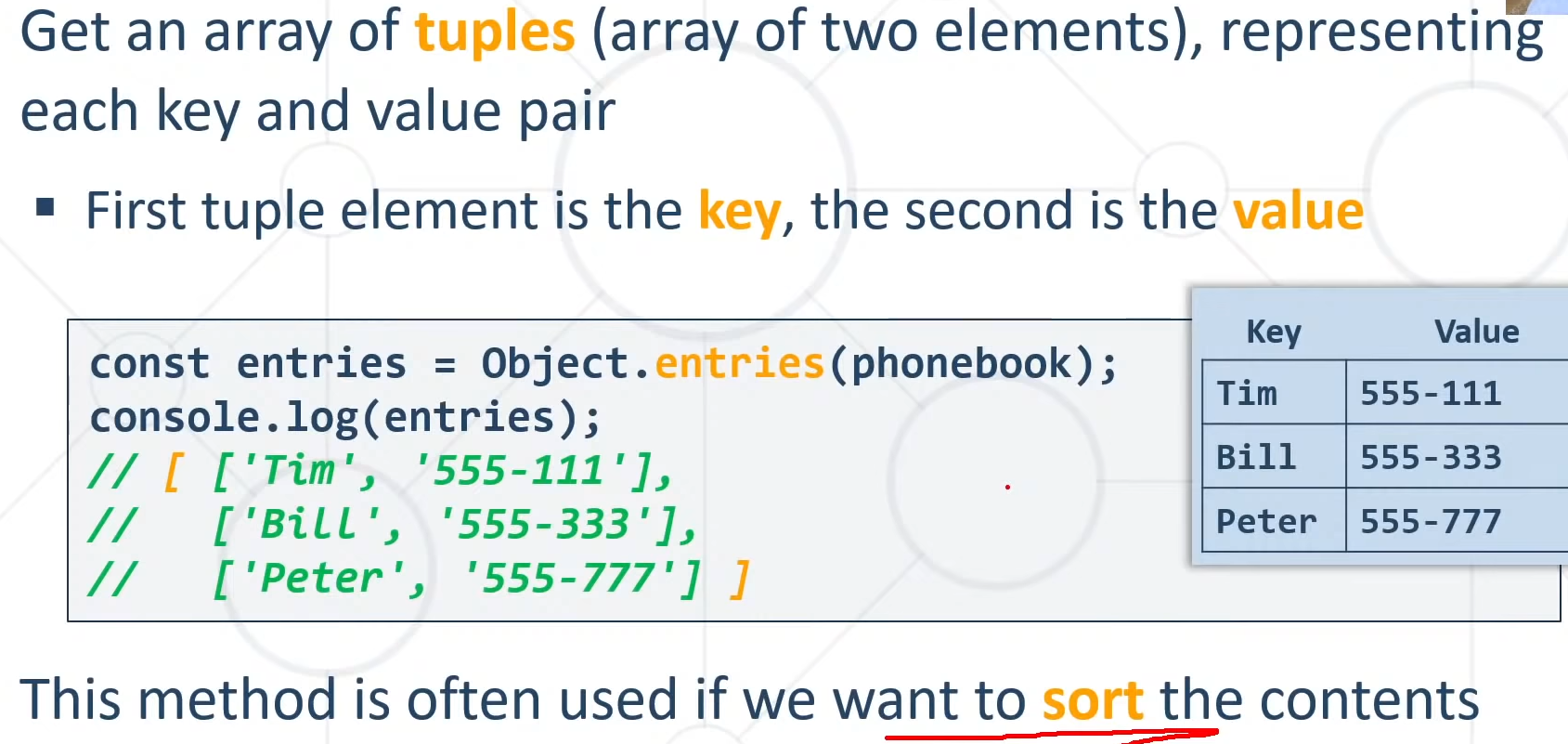


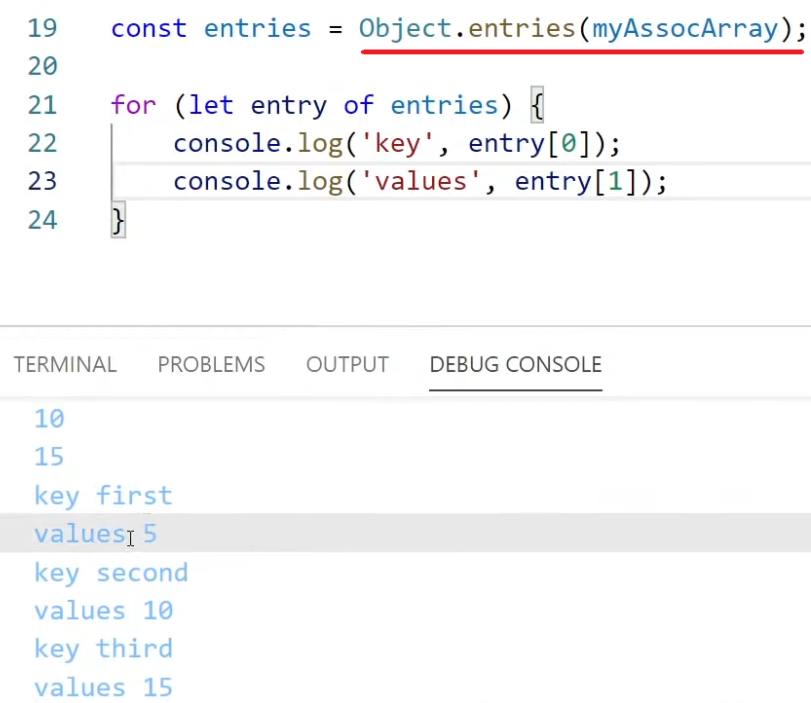
По същия начин можем да достъпим и Values, с разликата, че върнатият резултат няма да бъде масив, а итератор. Т.е. няма да можем да достъпваме конкретна стойност по индекс [].



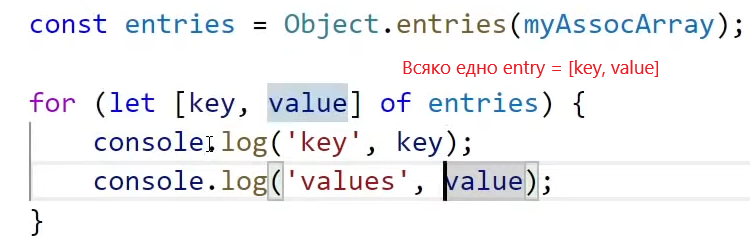
### Object.entries

Много често, най-вече с цел да можем да сотртираме, можем да изкарваме ключовете и стойностите на даден обект в отеделни масиви :



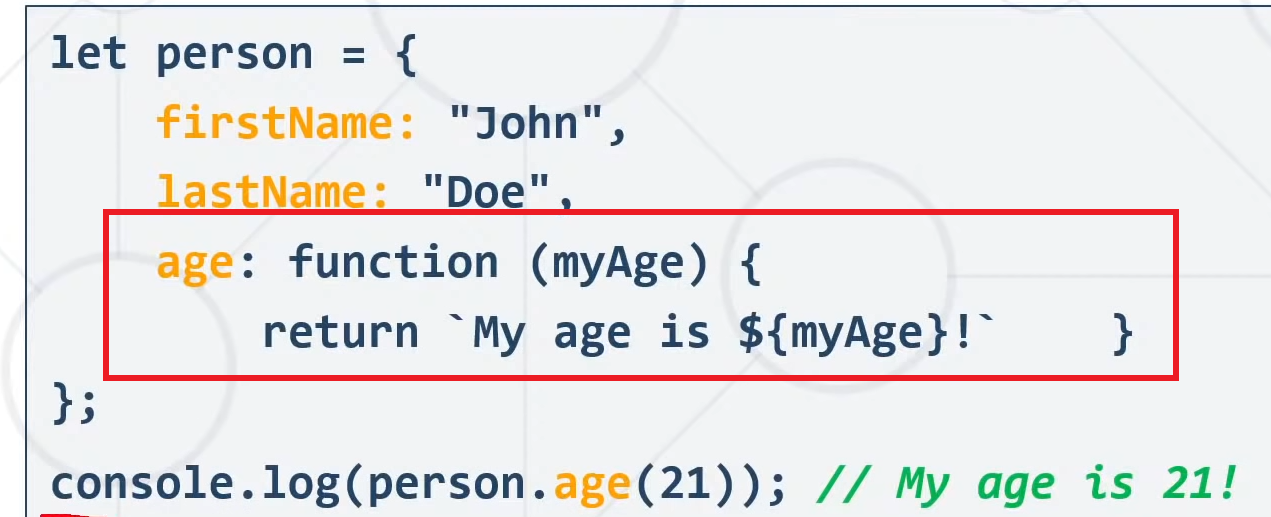


Най-често достъпваме ключове и стойности от Object.entries чрез деструктуриране на всяко едно entry :



## Methods and Context

Обектите могат да имат методи. Те на практика са функции, които се прикачат към ключовете по следния начин :

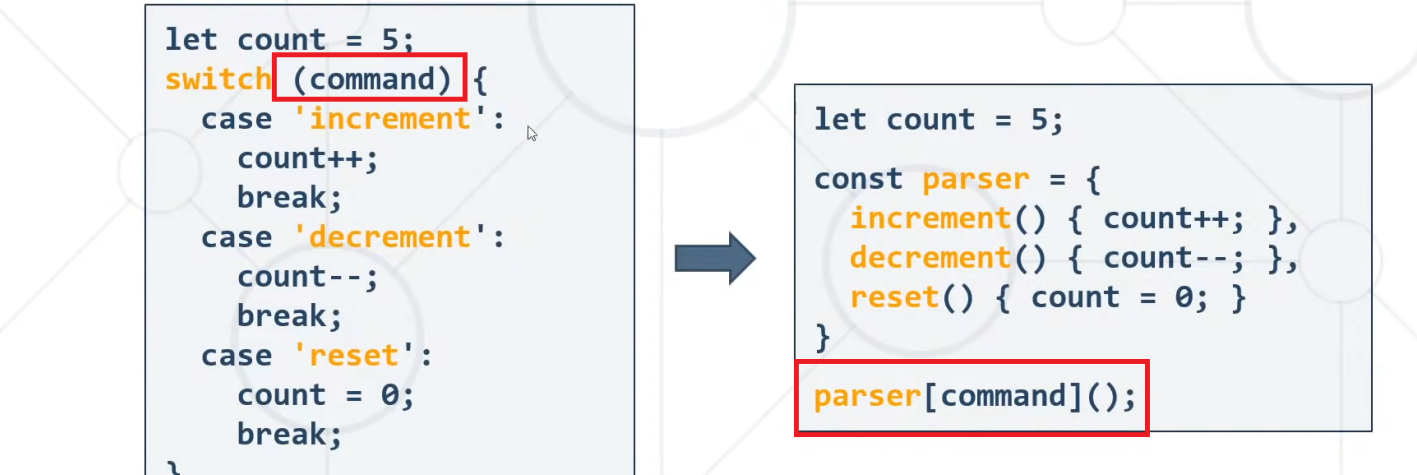


Извикват се като нормално свойство (Person.age())), като в случая трябва да подадем параметър – години.

Ако асоциативния масив е пълен с функции (методи) то той може да се разглежда като т.нар. **„function library”.** Такива примери са библиотеките Math, Object, Number etc, в които имаме много функции, които са групирани по някакъв общ признак на конкретния обект.

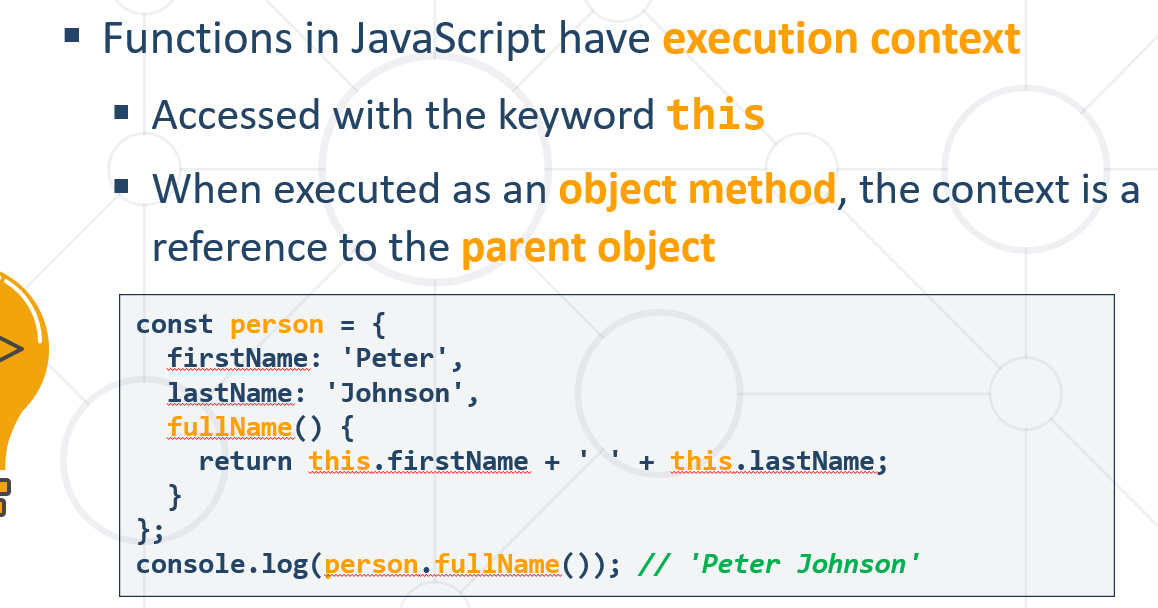
### Objects as switch replacement

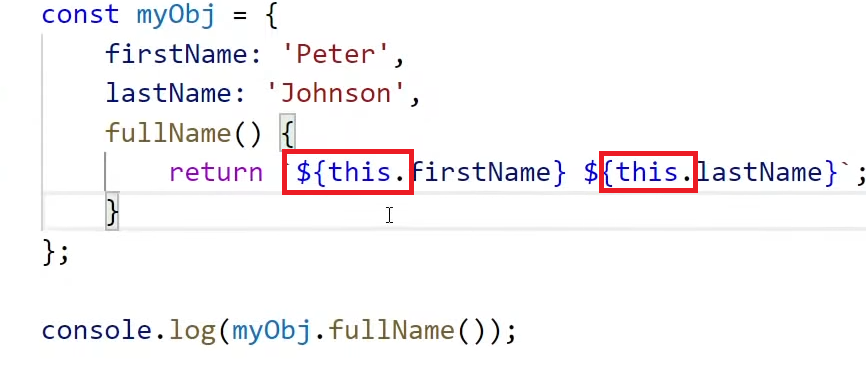
Поради наличието на обекти в JS, много рядко ще видим употребата на Switch case().



Вместо да пишем Switch, който **не може да се преизползва** и **е труден за Unit test**, е по-подходящо да **създадем обект**, който работи като **function library**. Накрая ще извикваме обекта, като все път подаваме [command], която ще идва от конзолата, и така ще извикваме функцията, която стои срещу нея.

## .this

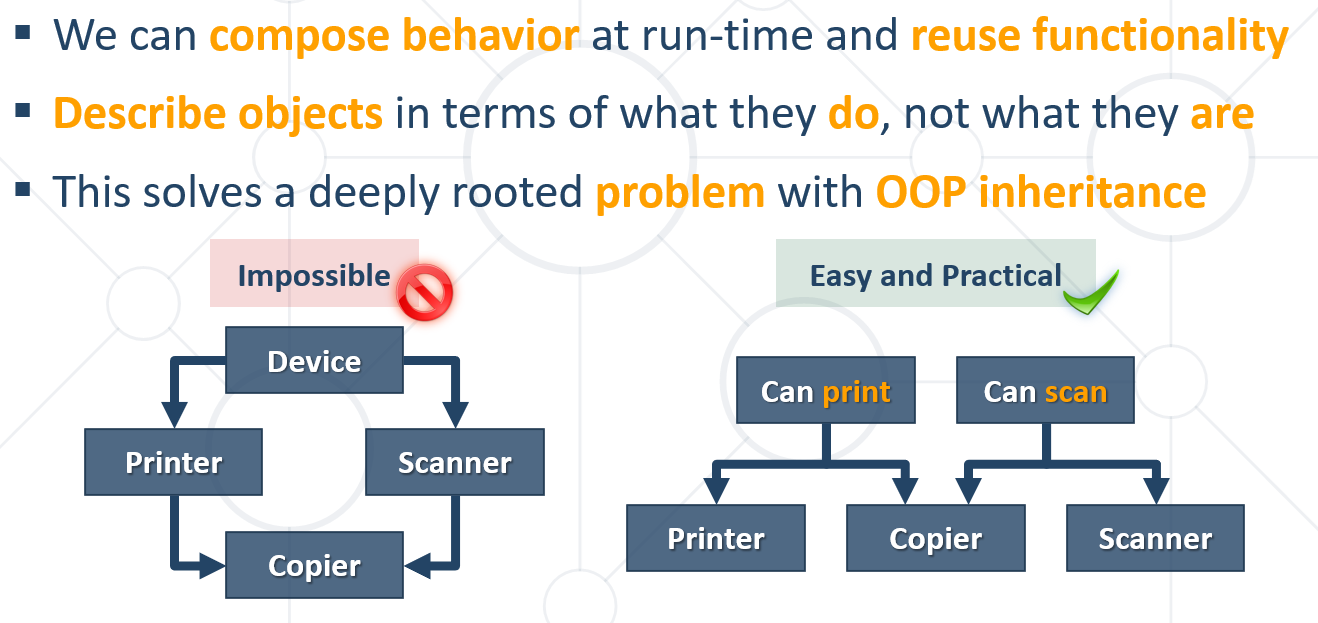




Както в горния пример, за да достъпим свойствата на обекта „firstName” и “lastName” с функцията fullName(), трябва да ги достъпим с .this, за да станат видими в scope-a на функцията.

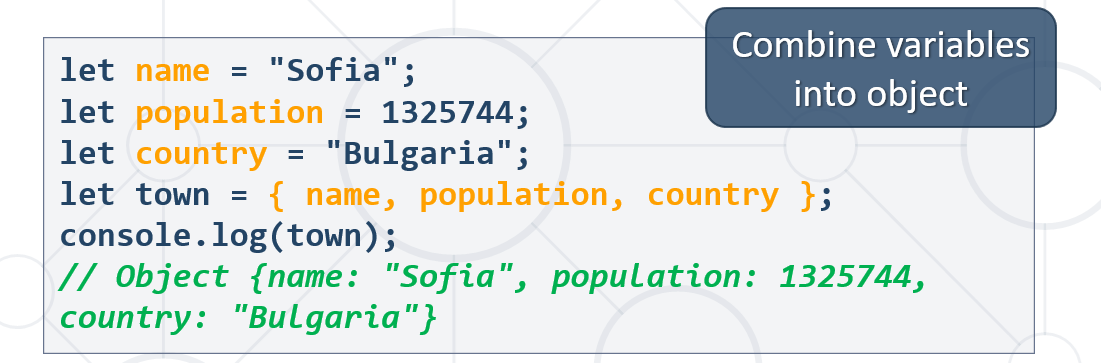
## Object Composition

Композирането на обекти в JS е супер силен инструмент, който се счита за по-добър дори от ООП концепцията. Под композиране на обект се разбира събирането на отделни свойства и функции в един комплексен обект.



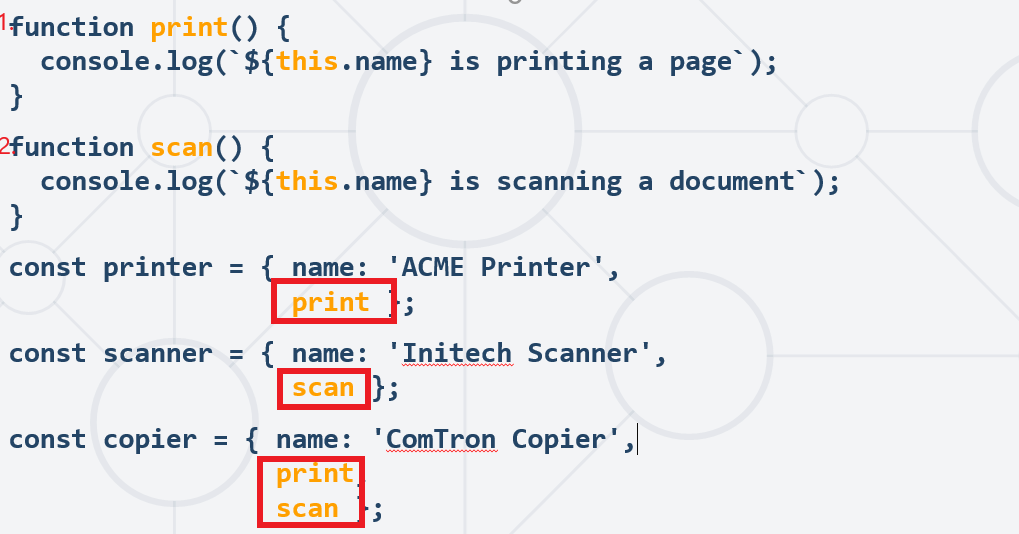
В JS ограниченията, които произхождат от „наследяването“ в ООП, с цел постигане на поли-морфизъм, се преодоляват, чрез **използването на функции**, които се **споделят между и от обектите.** Нещо като интерфейсите в Java. В горния пример при наследяването на Device имаме Printer and Scanner, като след това се явява нуждата да имплементираме и Copier,който обаче е едновременно Printer and Scanner. Това в JS се счита за затруднение на кода, което се преодолява с композиция. В случая в JS ще се създадат две функции – print and scan, които ще се имплементират в конструкторите на съотвените обекти – printer, copier and scanner. **С други думи в Java обектите отговарят на въпроса – какъв е обектът, докато в Js водещо е – какво може да прави ?**

Как композираме даден обект ?



Един пример е когато например създадем обекта town, който е съставен от name,population and country свойствата на обекта**.Така на практика създаваме обект в обекта.**

### Пример за композиране на обекти:



Имаме за задача да създадем три обекта машини – принтер, скенер и копирна машина. Принтерът трябв да може да принтира, скенерът – да сканира, а копирната машина – и двете неща. Това може да се постигне чрез композиция, която се изразява в следното :

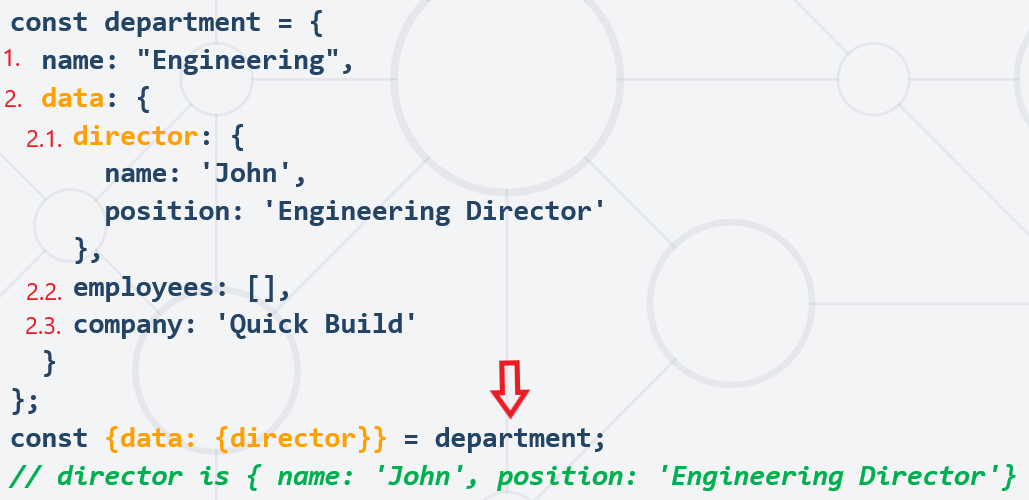
1) Създаване на функции – print() и scan();

2) Създаване на обекти – const printer, scanner и copier.

3) Прикачане на функциите към обектите, правейки ги техни методи.

По този начин ние „композираме“ обект – създаваме отделни парчета код (функции), които прикачаме към о

### Nested destructuring

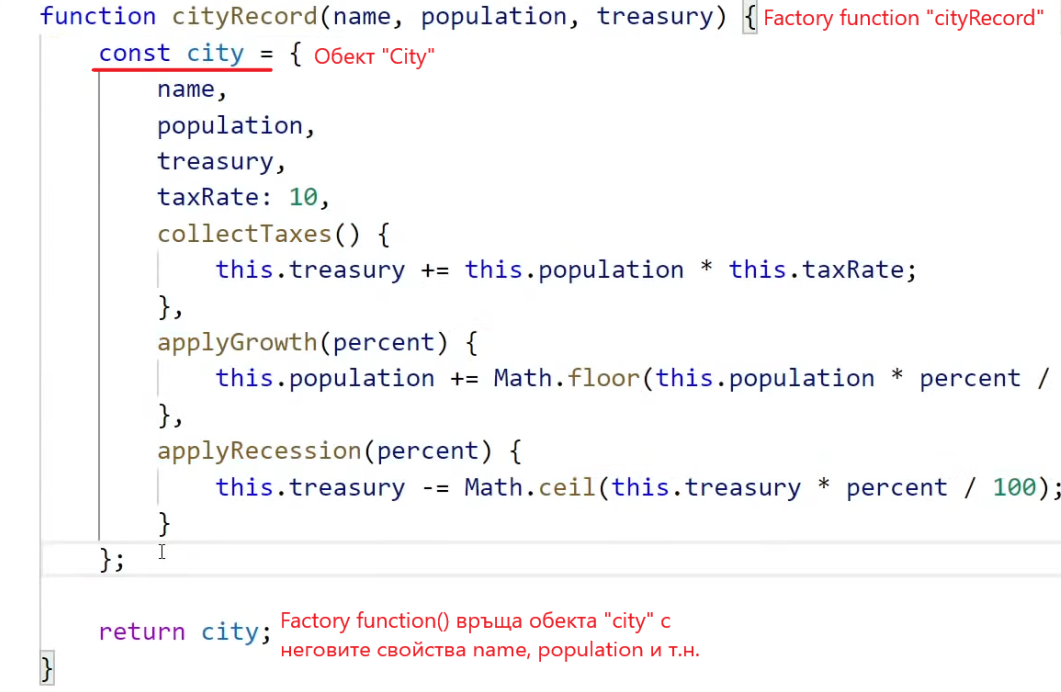


Вложено деструктуриране се получава когато искаме да достъпим някое от полетата на обект, което също само по себе си е обект. В горния случай имаме **обект „department”,** който се състои от **обект data**, състоящ се от **обект director**, както и още имаме **свойства employees и company;** За да достъпим обекта director, трябва да кажем : **дай ми от обекта department** обекта data {data: … , след което ми бръкни в data: и ми дай обекта {director} . Чрез {} достъпваме вложените обекти в един обект.

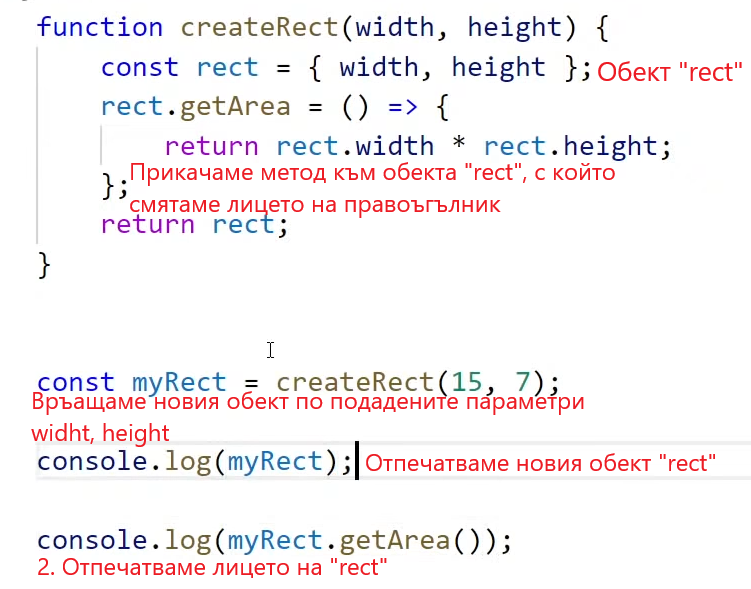
### Factory functions()

#### Пример 1:

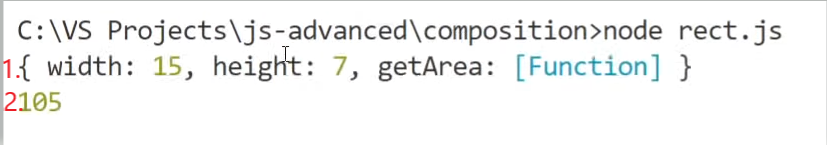
В практиката в JS рядко ще имаме случай в който създаваме директно някакви „хвърчащи“ обекти. Обикновено ги подаваме на т.нар. „factory functions”, които по зададени параметри връщат готови обекти. Един пример за factory function() е този по-долу :



#### Пример 2 :



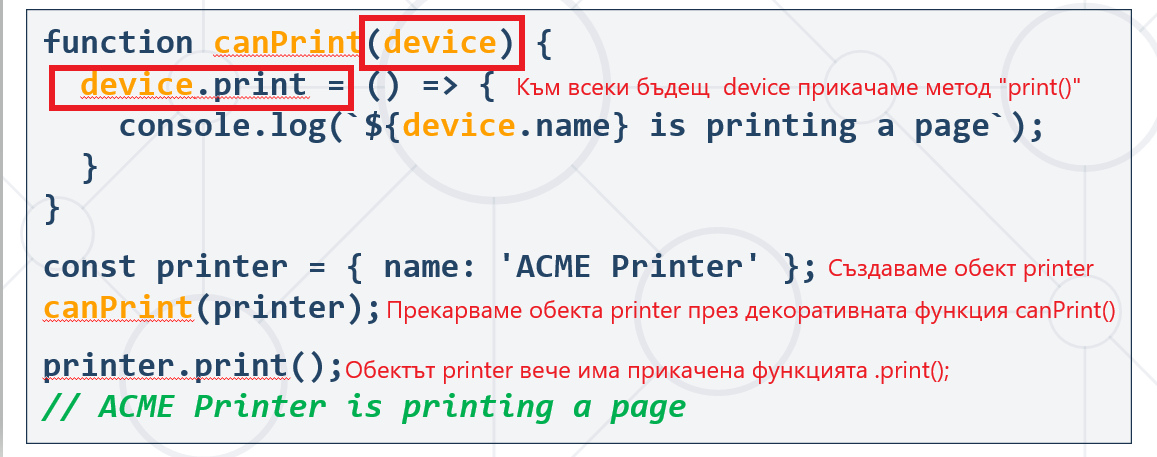
=>



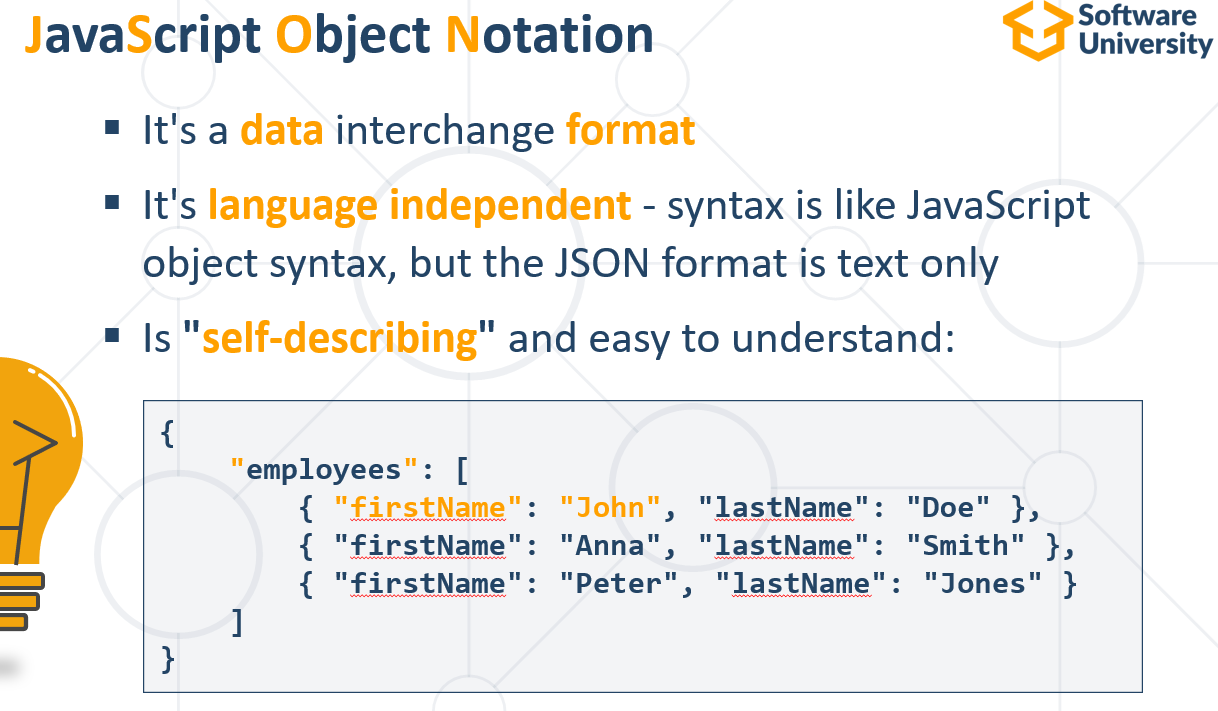
### Decorator functions

**Oще един начин да направим композиция на обект е с т.нар. „Декоративни функции**“. Докато при Factory() functions създавахме (композирахме) обект по дадени параметри, при decorator функциите **ние подаваме готов обект**, към който искаме да д**обавим някаква допълнителна функционалност** (метод). Като резултат имаме същия обект, към който сме добавили **нови свойства.**

#### Пример:



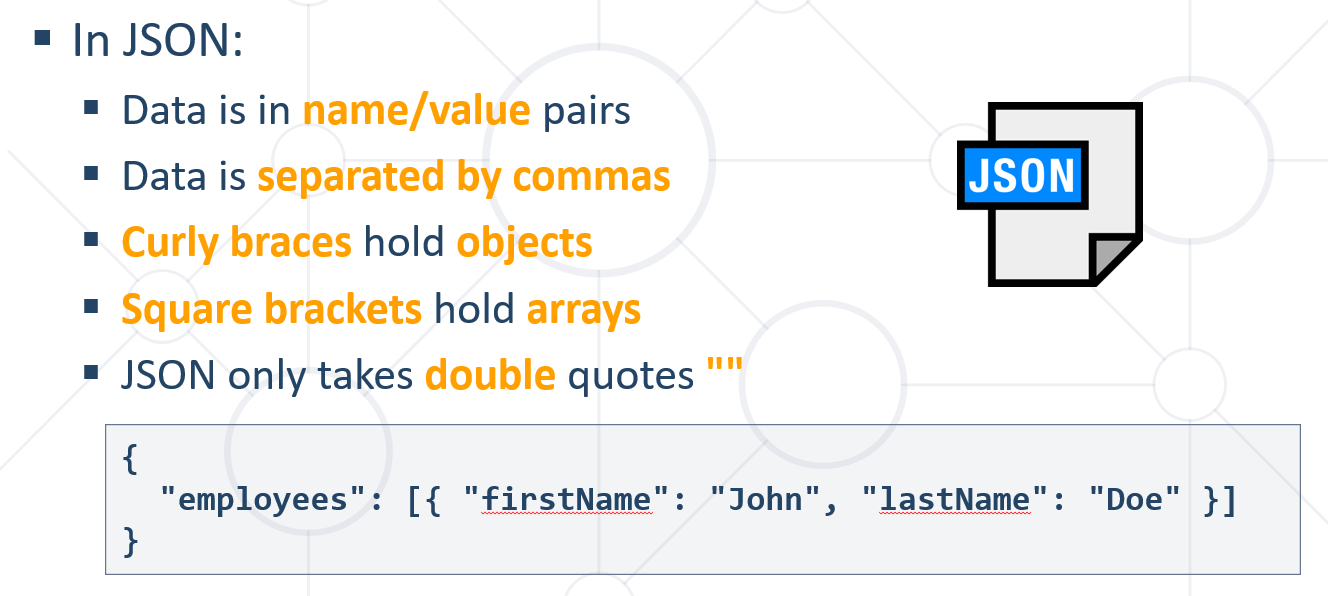
## JSON



JSON означава **J**ava **S**cript Objection Notation. Това е стандарт за обмен на информация и начин да серилиазираме обекти в JavaScript. JSON се използва във всички обекти, въпреки това, че в името му има JavaScript. Преди години стандартът е бил XML, който с времето се е доказал като не толкова добър вариант.

Syntax rules of JSON :

Много сходен е с този на JS, с изключение на “ “ за String :



За какво се използва JSON ?

1. В общия случай JSON се използва за четене на данни от web server-a, и за display-ване на дататата в web страниците.

2. JSON работи с два основни метода – JSON.parse() и JSON.stringify(). Първият превръща JSON (string) в обект, а втория прави обратното – конвертира обектите в JSON.

# II. Задачи

## 1. City Record

You will receive a city’s **name** (string), **population** (number), and **treasury** (number)as arguments, which you will need to set as **properties** of an **object** and **return** it.

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| 'Tortuga',  7000,  15000 | {  name: 'Tortuga',  population: 7000,  treasury: 15000  } |
| 'Santo Domingo',  12000,  23500 | {  name: 'Santo Domingo',  population: 12000,  treasury: 23500  } |

function solve(name, population, treasury) {

    const cityRecord = {

        name: name,

        population: population,

        treasury: treasury

    };

    return cityRecord;

}

console.log(solve(['Tortuga',

    7000,

    15000

]));

## 2. Town Population

You have been tasked to create a registry for different **towns** and their **population**.

### Input

The **input** comes as array of strings. Each element will contain data for a town and its population in the following format: "{townName} <-> {townPopulation}"

If you receive the same town twice, **you should add** the **given population** to the **current one**.

### Output

As **output**, you must print all the towns, and their population.

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| ['Sofia <-> 1200000',  'Montana <-> 20000',  'New York <-> 10000000',  'Washington <-> 2345000',  'Las Vegas <-> 1000000'] | Sofia : 1200000  Montana : 20000  New York : 10000000  Washington : 2345000  Las Vegas : 1000000 |
| ['Istanbul <-> 100000',  'Honk Kong <-> 2100004',  'Jerusalem <-> 2352344',  'Mexico City <-> 23401925',  'Istanbul <-> 1000'] | Istanbul : 101000  Honk Kong : 2100004  Jerusalem : 2352344  Mexico City : 23401925 |

function solve(arr) {

  const towns = {};

  let tuples = Object.entries(arr);

  for (let e of tuples) {

    const string = e[1];

    let [name, popStr] = string.split(" <-> ");

    let population = Number(popStr);

    if (!towns[name]) {

      towns[name] = population;

    } else {

      towns[name] += population;

    }

  }

  for (const key in towns){

    console.log(`${key} : ${towns[key]}`);

  }

}

solve(['Sofia <-> 1200000',

    'Montana <-> 20000',

    'New York <-> 10000000',

    'Washington <-> 2345000',

    'Las Vegas <-> 1000000']

);

## 3. City Taxes

*This task is an extension of Problem 1, you may use your solution from that task as a base.*

You will receive a city’s **name** (string), **population** (number), and **treasury** (number)as arguments, which you will need to set as **properties** of an **object** and **return** it. In addition to the input parameters, the object must have a property taxRate with initial value **10**, and three **methods** for managing the city:

* collectTaxes() **-** Increase **treasury** by population \* taxRate
* applyGrowth(percentage) **-** Increase population by **given percentage**
* applyRecession(percentage) **-** Decrease treasury by **given percentage**

Round down the values after each calculation.

### Input

Your solution will receive three valid parameters. The methods that expect parameters will be tested with valid input.

### Output

Return an object as described above. The methods of the object modify the object and don’t return anything.

|  |  |
| --- | --- |
| **Input** | **Output** |
| const city =  cityTaxes('Tortuga',  7000,  15000);  console.log(city); | {  name: 'Tortuga',  population: 7000,  treasury: 15000,  taxRate: 10,  collectTax: [Function: collectTax],  applyGrowth: [Function: applyGrowth],  applyRecession: [Function: applyRecession]  } |
| **Testing with code** | |
| **Input** | **Output** |
| const city =  cityTaxes('Tortuga',  7000,  15000);  city.collectTaxes();  console.log(city.treasury);  city.applyGrowth(5);  console.log(city.population); | 85000  7350 |

function cityRecord(name, population, treasury) {

    const city = {

        name: name,

        population: population,

        treasury: treasury,

        taxRate: 10,

        collectTaxes() {

            this.treasury += this.population \* this.taxRate;

        },

        applyGrowth(percent) {

            this.population += Math.floor(percent / 100 \* this.population);

        },

        applyRecession(percent) {

            this.treasury -= Math.ceil(this.treasury \* percent / 100);

        }

    };

    return city;

}

const result = cityRecord(

    'Tortuga',

    7000,

    15000);

console.log(result);

## 4. Object Factory

Create a function that can compose objects by copying functions from a given library of functions. You will receive **two** **parameters** – a **library** of functions as an associative array (object) and an **array of orders**, represented as objects**.** You must **return** a new array – the fulfilled orders.

The **first parameter** will be an object where each property is a **function**. You will use this **library of functions** to compose new objects.

The **second parameter** is an **array of orders**. Each order is an **object** with the following shape:

{

template: [Object],

parts: string[]

}

The **template** is an object that must be **copied**. The **parts array** contains the names of **required functions** as **strings**.

You must **create and return a new array**, by fulfilling all orders from the **orders array**. To fulfill an order, create a copy of the object’s template and then add to it all functions, listed in the **parts array** of the order, by taking them from the **function library** (first parameter to your solution).

### Input

You will receive two parameters:

* library – an object
* orders – an array of objects

### Output

Your solution must **return an array** of objects.

### Example

|  |
| --- |
| **Input** |
| const library = {  print: function () {  console.log(`${this.name} is printing a page`);  },  scan: function () {  console.log(`${this.name} is scanning a document`);  },  play: function (artist, track) {  console.log(`${this.name} is playing '${track}' by ${artist}`);  },  };  const orders = [  {  template: { name: 'ACME Printer'},  parts: ['print']  },  {  template: { name: 'Initech Scanner'},  parts: ['scan']  },  {  template: { name: 'ComTron Copier'},  parts: ['scan', 'print']  },  {  template: { name: 'BoomBox Stereo'},  parts: ['play']  },  ];  const products = factory(library, orders);  console.log(products); |
| **Output** |
| [  {  name: 'ACME Printer',  print: [Function: print]  },  {  name: 'Initech Scanner',  scan: [Function: scan]  },  {  name: 'ComTron Copier',  scan: [Function: scan],  print: [Function: print]  },  {  name: 'BoomBox Stereo',  play: [Function: play]  },  ] |

function factory(library, orders) {

    const result = [];

    //iterate over orders

    for (let order of orders) {

        //copy order termplate

        const composed = Object.assign({}, order.template);

        //compose methods by part list

        for (let part of order.parts) {

            composed[part] = library[part];

        }

        result.push(composed);

    }

    return result;

}

const library = {

    print: function () {

        console.log(`${this.name} is printing a page`);

    },

    scan: function () {

        console.log(`${this.name} is scanning a document`);

    },

    play: function (artist, track) {

        console.log(`${this.name} is playing '${track}' by ${artist}`);

    },

};

const orders = [

    {

        template: { name: 'ACME Printer' },

        parts: ['print']

    },

    {

        template: { name: 'Initech Scanner' },

        parts: ['scan']

    },

    {

        template: { name: 'ComTron Copier' },

        parts: ['scan', 'print']

    },

    {

        template: { name: 'BoomBox Stereo' },

        parts: ['play']

    },

];

const products = factory(library, orders);

console.log(products);

const player = products[3];

player.play('Madonna', "Forever");

## 5. Assembly Line

Create a function that **returns** a **library of decorator functions**. They can be used to **compose** different functionality in a **car object** that they receive as argument.

Your solution must **return an object**, containing **three decorator functions**:

hasClima – compose air conditioning controls into the passed in object. This function takes an **object as parameter** and adds to it the following properties:

* temp – **number** with default value **21**;
* tempSettings – **number** with default value **21**;
* adjustTemp – **function** which takes **no arguments**. If temp is less than tempSettings, this function adds 1 to temp. If temp is more than tempSettings, it decreases temp by 1. If temp and tempSettings are equal, the function does nothing.

hasAudio – compose audio player functionality into the passed in object. This function takes an **object as parameter** and adds to it the following properties:

* currentTrack – **object** with properties name (string) and artist (string). Default value is null;
* nowPlaying – **function**, which **prints** on the console the text "Now playing '{currentTrack.name}' by ${currentTrack.artist}", where name and artist are properties of the currentTrack object. If currentTrack is null, this function does nothing.

hasParktronic – compose parking aid functionality into the passed in object. This function takes an **object as parameter** and adds to it the following properties:

* checkDistance – **function**, which takes a **single argument** distance (number) and **prints** a message on the console, depending on its value:

distance < 0.1 – "Beep! Beep! Beep!"

0.1 <= distance < 0.25 – "Beep! Beep!"

0.25 <= distance < 0.5 – "Beep!"

In any other case, print an **empty string**.

### Input

Your **solution** will receive **no arguments**. All the methods in the returned library must take an **object as argument**. Any methods that you compose into this object must meet the input requirements listed in the description above.

### Output

Your **solution** must **return an object** containing the **three decorators** described above.

### Example

|  |  |
| --- | --- |
| **Setup** | |
| const assemblyLine = createAssemblyLine();  const myCar = {  make: 'Toyota',  model: 'Avensis'  }; | |
| **Input** | **Output** |
| assemblyLine.hasClima(myCar);  console.log(myCar.temp);  myCar.tempSettings = 18;  myCar.adjustTemp();  console.log(myCar.temp); | 21  20 |
| **Input** | **Output** |
| assemblyLine.hasAudio(myCar);  myCar.currentTrack = {  name: 'Never Gonna Give You Up',  artist: 'Rick Astley'  };  myCar.nowPlaying(); | Now playing 'Never Gonna Give You Up' by Rick Astley |
| **Input** | **Output** |
| assemblyLine.hasParktronic(myCar);  myCar.checkDistance(0.4);  myCar.checkDistance(0.2); | Beep!  Beep! Beep! |
| **Input** | **Output** |
| console.log(myCar); | {  make: 'Toyota',  model: 'Avensis',  temp: 20,  tempSettings: 18,  adjustTemp: [Function],  currentTrack: {  name: 'Never Gonna Give You Up',  artist: 'Rick Astley'  },  nowPlaying: [Function],  checkDistance: [Function]  } |

## From JSON to HTML Table

You’re tasked with creating an HTML table of students and their scores. You will receive a single string representing an **array of objects**, the **table’s headings** should be equal to the **object’s keys**, while **each object’s values** should be a **new entry** in the table. Any **text values** in an object should be **escaped**, in order to avoid introducing dangerous code into the HTML.

### Input

The **input** comes a **single string argument** (the array of objects).

### Output

The **output** should be printed on the console – for each **entry** **row** in the input print the **object** **representing** **it**.

### Note:

Object’s **keys** will always be the **same.** Check more information for the **HTML Entity** [**here**](https://developer.mozilla.org/en-US/docs/Glossary/Entity)**.**

### HTML

You are provided with an HTML file to test your table in the browser.

|  |
| --- |
| index.html |
| <!DOCTYPE **html**> <**html lang="en"**> <**head**>  <**meta charset="UTF-8"**>  <**title**>FromJSONToHTMLTable</**title**>  <**style**>  **table**,**th**{  **border**: **groove**;  **border-collapse**: **collapse**;  }  **td**{  **border**: 1**px solid black**;  }  **td**,**th**{  **padding**: 5**px**;  }  </**style**> </**head**> <**body**>  <**div id="wrapper"**>  </**div**>  <**script**>  **function** *fromJSONToHTMLTable*(input){  *//Write your code here* }  **window**.onload = **function**(){  **let** container = **document**.getElementById(**'wrapper'**);  container.**innerHTML** = *fromJSONToHTMLTable*([**'[{"Name":"Stamat","Price":5.5},{"Name":"Rumen","Price":6}]'**]);  };  </**script**> </**body**> </**html**> |

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| '[{"Name":"Stamat",  "Score":5.5},  {"Name":"Rumen",  "Score":6}]' | <table>  <tr><th>Name</th><th>Score</th></tr>  <tr><td>Stamat</td><td>5.5</td></tr>  <tr><td>Rumen</td><td>6</td></tr>  </table> |
| '[{"Name":"Pesho",  "Score":4,  " Grade":8},  {"Name":"Gosho",  "Score":5,  " Grade":8},  {"Name":"Angel",  "Score":5.50,  " Grade":10}]' | <table>  <tr><th>Name</th><th>Score</th><th>Grade</th></tr>  <tr><td>Pesho</td><td>4</td><td>8</td></tr>  <tr><td>Gosho</td><td>5</td><td>8</td></tr>  <tr><td>Angel</td><td>5.5</td><td>10</td></tr>  </table> |

# III. Термини

## 1. JSON

## 2. Обекти в JS

## 3. Асоциативни обекти в JS

## 4. Композиция и предимстава й пред OOP

## 5. Factory() function

## 6. Decorator() function

## 7. Плитко и дълбоко копиране на обект