

# Objects & Composition



**SoftUni Team**  
**Technical Trainers**



**SoftUni**



**Software University**

<https://softuni.bg>

# Table of Contents

1. Objects in JavaScript
2. Objects as Associative Arrays
3. Methods and Context
4. Object Composition
5. JSON





sli.do

#js-advanced



# Objects in JavaScript

Definition and Properties

# What is an Object?

- An object is a **collection of fields**, called **properties**
- A property is an association between a name (or **key**) and a **value**
- Objects are a **reference data type**
- In JavaScript they are created with an **object literal**:

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



# Object Properties

- A **property** of an object can be explained as a **variable** that is **attached** to the object
- Object properties are the same as **ordinary variables**, and can hold **any data type** and be **reassigned**



Property Name	Property Value
firstName	John
lastName	Doe
age	50

# Problem: City Record

- Receive three parameters **name**, **population** and **treasury**
- Create and **return** an **object** with properties of the same names

'Tortuga', 7000, 15000



```
{ name: 'Tortuga',  
  population: 7000,  
  treasury: 15000 }
```

'Santo Domingo', 12000, 23500



```
{ name: 'Santo Domingo',  
  population: 12000,  
  treasury: 23500 }
```

# Solution: City Record

```
function createRecord(name, population, treasury) {  
  const city = {};  
  
  city.name = name;  
  city.population = population;  
  city.treasury = treasury;  
  
  return city;  
}
```

```
function createRecord(name, population, treasury) {  
  return {  
    name,  
    population,  
    treasury  
  };  
}
```



# Assigning and Accessing Properties

- Simple **dot-notation**

```
const person = { name: 'Peter' };  
console.log(person.name); // Peter
```

- **Bracket-notation** (indexing operator)
  - Required if the key contains a **special character**

```
person['job-title'] = 'Trainer';  
console.log(person['job-title']) // Trainer  
console.log(person.job-title)    // ReferenceError
```

- Brackets can be used with keys as **string variables**



# Assigning and Accessing Properties (2)

- Properties can be **added** during run-time

```
const person = { name: 'Peter' };  
person.age = 21; // { name: 'Peter', age: 21 }  
console.log(person.age); // 21
```

- **Unassigned** properties of an object are **undefined**

```
const person = { name: 'Peter' };  
console.log(person.lastName); // undefined
```



- **"Dive into"** an **object** and extract properties by name
- Can be used to get **multiple** property values

```
const department = {  
  name: 'Engineering',  
  director: 'Ted Thompson',  
  employeeCount: 25  
};  
  
const { name, employeeCount } = department;  
console.log(name, employeeCount); // 'Engineering' 25
```

# Destructuring Syntax

```
const obj = { a: 1, b: 2, c: 3 };

const { c, ...props } = obj;

const modifiedObj = {
  ...props,
  c: 12,
};

console.log(obj); // { a: 1, b: 2, c: 3, d: 4 }
console.log(modifiedObj); // { a: 1, b: 2, c: 3, d: 12 }
```

# Deleting Properties

```
const person = {  
  name: 'Peter',  
  age: 21,  
  ['job-title']: 'Trainer'  
}  
// Object {name: 'Peter', age: 21, 'job-title': 'Trainer' }  
  
delete person.age;  
// Object {name: 'Peter', 'job-title': 'Trainer' }  
  
console.log(person.age) // undefined
```

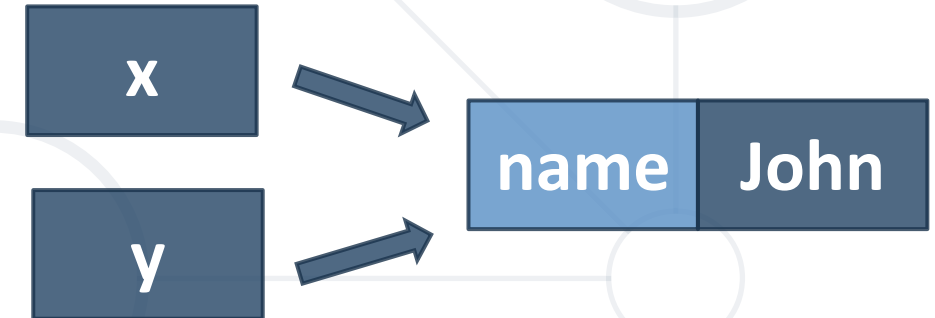
# Object References

- Variables holding **reference** data types contain the **memory address** (reference) of the data
- Copies** of the reference point to the **same data**

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

```
let y = x;
```

```
y.name = 'Steve';  
console.log(x.name); // Steve
```



- Two variables, **two distinct objects** with the same properties

```
const fruit = {name: 'apple'};  
const fruitbear = {name: 'apple'};  
fruit == fruitbear; // false  
fruit === fruitbear; // false
```

- Two variables, a **single object**

```
const fruit = { name: 'apple' };  
const fruitbear = fruit;  
// Assign a copy of the fruit reference to fruitbear  
fruit == fruitbear; // true  
fruit === fruitbear; // true
```



# Objects as Associative Arrays

Property Lookup and Iteration



- Objects can serve the role of **associative arrays** in JavaScript
  - The keys (property names) are **string indexes**
  - Values are **associated** to a key
  - All values should be of the **same type**

Key	Value
John Smith	+1-555-8976
Lisa Smith	+1-555-1234
Sam Doe	+1-555-5030

# For... in Loop

- **for ... in** – iterates over all **enumerable** properties

```
const obj = { a: 1, b: 2, c: 3 };
```

```
for (const key in obj) {  
  console.log(key);  
  console.log(obj[key]);  
}
```

```
// a
```

```
// 1
```

```
// b
```

```
// 2
```

```
// c
```

```
// 3
```



# Object Keys and Values

- Obtain an **array** of all **keys** or **values** in an object:

```
const phonebook = { 'Tim': '555-111',  
                    'Bill': '555-333',  
                    'Peter': '555-777' };  
  
const keys = Object.keys(phonebook);  
console.log(keys);  
// ['Tim', 'Bill', 'Peter']
```

Key	Value
Tim	555-111
Bill	555-333
Peter	555-777

```
const values = Object.values(phonebook);  
console.log(values);  
// ['555-111', '555-333', '555-777']
```

- Get an array of **tuples** (array of two elements), representing each key and value pair
  - First tuple element is the **key**, the second is the **value**

```
const entries = Object.entries(phonebook);  
console.log(entries);  
// [ ['Tim', '555-111'],  
//   ['Bill', '555-333'],  
//   ['Peter', '555-777'] ]
```

Key	Value
Tim	555-111
Bill	555-333
Peter	555-777

- This method is often used if we want to **sort** the contents

# Problem: Town Population

- Receive an array of string in format:  
**"{townName} <-> {townPopulation}"**
- Create a registry for each town
- If a town repeats, add the population to the existing value

```
['Istanbul <-> 100000',  
'Honk Kong <-> 2100004',  
'Jerusalem <-> 2352344',  
'Mexico City <-> 23401925',  
'Istanbul <-> 1000']
```



```
Istanbul : 101000  
Honk Kong : 2100004  
Jerusalem : 2352344  
Mexico City : 23401925
```

# Solution: Town Population

```
function townPopulation(townsArr) {
  const towns = {};

  for (let townAsString of townsArr) {
    // Split each string into 'name' and 'population' using ' <-> ' as the separator
    let [name, population] = townAsString.split(' <-> ');
    population = Number(population);

    // Check if the town 'name' already exists in the 'towns' object
    if (towns[name] !== undefined) {
      // If the town exists, add the 'population' to the existing value in 'towns'
      population += towns[name];
    }

    // Update the 'towns' object with the new 'population' value for the 'name'
    towns[name] = population;
  }

  for (let town in towns) {
    console.log(`${town} : ${towns[town]}`);
  }
}
```




# Methods and Context

Combine Data with Behavior

# Object Methods

- Objects can also have **methods**
- Methods are **actions** that can be performed on objects
- Methods are stored in **properties** as **function** definitions



```
let person = {  
  firstName: "John",  
  lastName: "Doe",  
  printAge: function (myAge) {  
    return `My age is ${myAge}!`  
  }  
};  
console.log(person.printAge(21)); // My age is 21!
```



# Objects as Function Libraries

- Related functions may be **grouped** in an object
- The object serves as a **function library**
  - Similar to built-in libraries like **Math**, **Object**, **Number**, etc.

```
// sorting helper  
const compareNumbers = {  
  ascending: (a, b) => a - b;  
  descending: (a, b) => b - a;  
};
```

- This technique is often used to **expose public API** in a module

# Objects as **switch** replacement

- You will **almost never** see **switch** used in JS code
- **Named methods** are used instead

```
let count = 5;  
switch (command) {  
  case 'increment':  
    count++;  
    break;  
  case 'decrement':  
    count--;  
    break;  
  case 'reset':  
    count = 0;  
    break;  
}
```



```
let count = 5;  
  
const parser = {  
  increment() { count++; },  
  decrement() { count--; },  
  reset() { count = 0; }  
}  
  
parser[command]();
```

**Shorter syntax** for  
object methods

# Review: Cooking by Numbers

- Attempt to solve this problem from **previous exercises**, using **objects**, instead of **conditional statements**



# Accessing Object Context

- Functions in JavaScript have **execution context**
  - Accessed with the keyword **this**
  - When executed as an **object method**, the context is a reference to the **parent object**



```
const person = {
  firstName: 'Peter',
  lastName: 'Johnson',
  fullName() {
    return this.firstName + ' ' + this.lastName;
  }
};
console.log(person.fullName()); // 'Peter Johnson'
```

# Function Execution Context

- Execution context can be **changed** at run-time
- If a function is **executed outside** of its parent object, it will **no longer** have access to the object's content



```
const getFullName = person.fullName;
console.log(getFullName()); // 'undefined undefined'
const anotherPerson = { firstName: 'Bob',
                        lastName: 'Smith' };
anotherPerson.fullName = getFullName;
console.log(anotherPerson.fullName()); // 'Bob Smith'
```

- **Further lessons** will explore more **context features**!

# Problem: City Taxes

- Extend **Problem 1: City Record**
  - Add property **taxRate** with initial value **10**
  - Add **methods**:
    - collectTaxes()** increase **treasury** by (**population** \* **taxRate**)
    - applyGrowth(percent)** increase **population** by percentage
    - applyRecession(percent)** decrease **treasury** by percentage
  - All values must be **rounded down** after calculation

# Solution: City Taxes

```
function createRecord(name, population, treasury) {  
  return {  
    name, population, treasury,  
    taxRate: 10,  
    collectTaxes() {  
      this.treasury += this.population * this.taxRate;  
    },  
    applyGrowth(percent) {  
      this.population += Math.floor(this.population * percent / 100);  
    },  
    applyRecession(percent) {  
      this.treasury -= Math.floor(this.treasury * percent / 100);  
    },  
  };  
}
```



# Object Composition

Creating Complex Objects from Simple Pieces



# What is Object Composition?

- **Combining** simple objects into more **complex ones**

```
let student = {  
  firstName: 'Maria',  
  lastName: 'Lopez',  
  age: 22,  
  location: { lat: 42.698, lng: 23.322 }  
}  
console.log(student);  
console.log(student.location.lat);
```

- **Composition** is a powerful technique for **code reuse**
- It can be considered **superior** to **OOP inheritance**

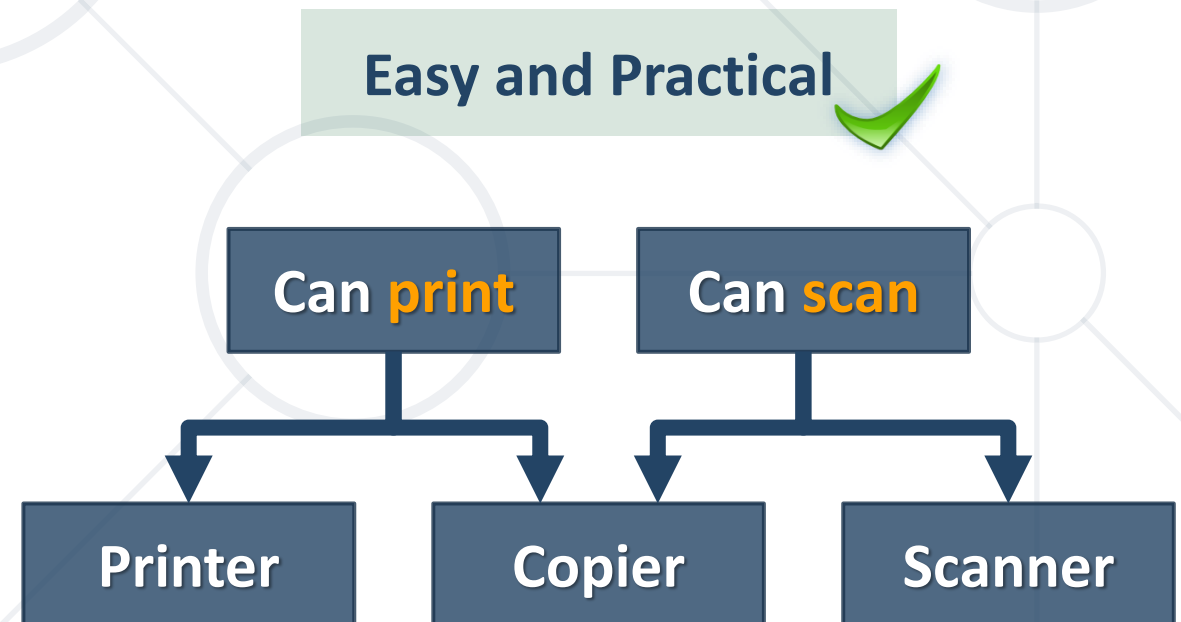
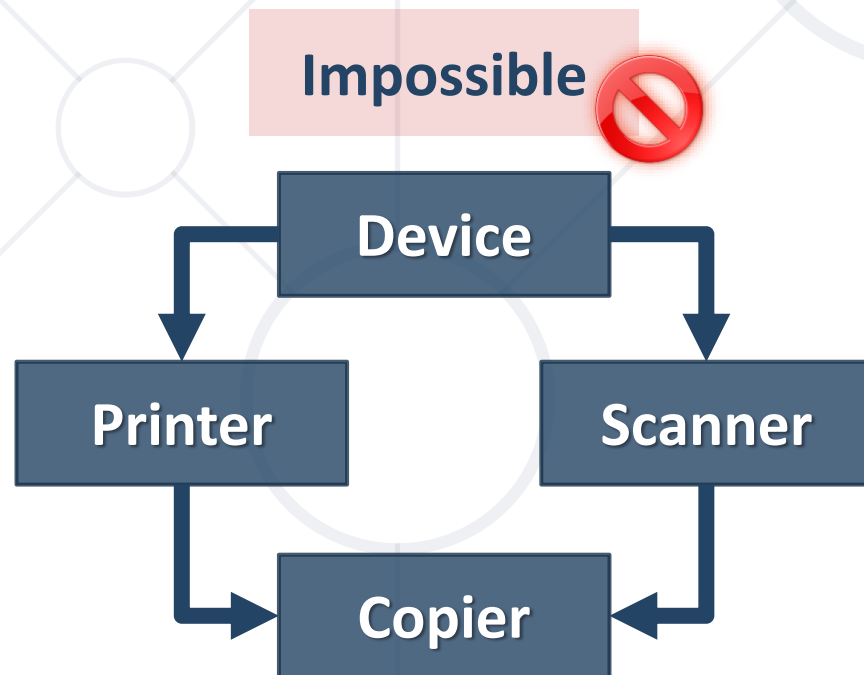


Combine variables  
into object

```
let name = "Sofia";  
let population = 1325744;  
let country = "Bulgaria";  
let town = { name, population, country };  
console.log(town);  
// Object {name: "Sofia", population: 1325744,  
country: "Bulgaria"}
```

```
town.location = { lat: 42.698, lng: 23.322 };  
console.log(town); // Object {..., location: Object}
```

- We can **compose behavior** at run-time and **reuse functionality**
- **Describe objects** in terms of what they **do**, not what they **are**
- This solves a deeply rooted **problem** with **OOP inheritance**




# Composing Objects with Behavior

```
function print() {  
  console.log(`${this.name} is printing a page`);  
}  
  
function scan() {  
  console.log(`${this.name} is scanning a document`);  
}  
  
const printer = { name: 'ACME Printer',  
                  print };  
  
const scanner = { name: 'Initech Scanner',  
                  scan };  
  
const copier = { name: 'ComTron Copier',  
                 print,  
                 scan };
```

# Factory Functions With Reference

- Functions that **compose** and **return** objects



```
function createRect(width, height) {  
  const rect = { width, height };  
  rect.getArea = () => {  
    return rect.width * rect.height;  
  };  
  return rect;  
}
```

- Creating methods with object references can **avoid** the pitfalls of using **this**

# Problem: Object Factory

- Receive two parameters **library** (object) and **orders** (array)  
**library** associative array of functions  
**orders** array of object with shape:  

```
{ template: <object>, parts: <string array> }
```
- For every **order**, create an object by copying its **template** and composing into it the required functions listed in its **parts**
- See next slide for **examples**

# Examples: Object Factory

library
<pre>{   doA: () =&gt; { /* ... */ },   doB: () =&gt; { /* ... */ },   doC: () =&gt; { /* ... */ } }</pre>

orders
<pre>[   { template: { id: 'first' },     parts: [ 'doB' ] },   { template: { id: 'second' },     parts: [ 'doA', 'doC' ] } ]</pre>



<pre>[   {     id: 'first',     doB: [Function: doB]   },   {     id: 'second',     doA: [Function: doA],     doC: [Function: doC]   } ]</pre>
------------------------------------------------------------------------------------------------------------------------------------------------

# Solution: Object Factory

```
function factory(library, orders) {  
  const result = [];  
  
  for (let order of orders) {  
    // Create a copy of the 'template' object from the current order  
    const current = Object.assign({}, order.template);  
    for (let part of order.parts) {  
      current[part] = library[part];  
    }  
    result.push(current);  
  }  
  
  return result;  
}
```



- Functions that **add** new **data** and **behavior** to objects

```
function canPrint(device) {  
  device.print = () => {  
    console.log(` ${device.name} is printing a page`);  
  }  
}  
  
const printer = { name: 'ACME Printer' };  
canPrint(printer);  
printer.print();  
// ACME Printer is printing a page
```

- The object reference is **embedded** – using **this** is not required



# Live Demonstration

Lab Problem 5



**JSON**

JavaScript Object Notation

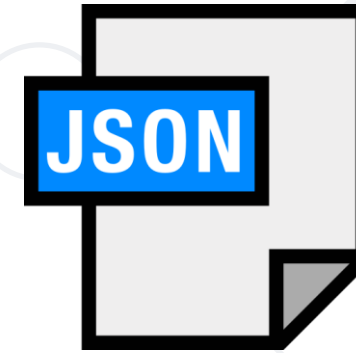
# JavaScript Object Notation

- It's a **data** interchange **format**
- It's **language independent** - syntax is like JavaScript object syntax, but the JSON format is text only
- Is "**self-describing**" and easy to understand:



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

- In JSON:
  - Data is in **name/value** pairs
  - Data is **separated by commas**
  - **Curly braces** hold **objects**
  - **Square brackets** hold **arrays**
  - JSON only takes **double** quotes **""**



```
{  
  "employees": [{ "firstName": "John", "lastName": "Doe" }]  
}
```

- A common use of JSON is to **read data from a web server**, and **display the data on a web page**
- Use the JavaScript built-in function **JSON.parse()** to convert the JSON format into a JavaScript object:

```
let data = '{ "manager":{"firstName":"John","lastName":"Doe"} }';  
let obj = JSON.parse(data);  
console.log(obj.manager.lastName) // Doe
```

- Use **JSON.stringify()** to convert objects into a string:

```
let obj = { name: "John", age: 30, city: "New York" };  
let myJSON = JSON.stringify(obj);  
console.log(myJSON); // {"name": "John", "age": 30, "city": "New York"}
```

- You can do the same for **arrays**

```
let arr = [ "John", "Peter", "Sally", "Jane" ];  
let myJSON = JSON.stringify(arr);  
console.log(myJSON); // ["John", "Peter", "Sally", "Jane"]
```

- **Format** the string with **indentation** for readability

```
let myJSON = JSON.stringify(arr, null, 2);
```

# Problem: From JSON to HTML Table

- Read a **JSON string**, holding array of JS objects
- Print the objects as **HTML table** like shown below

```
[{"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>
```



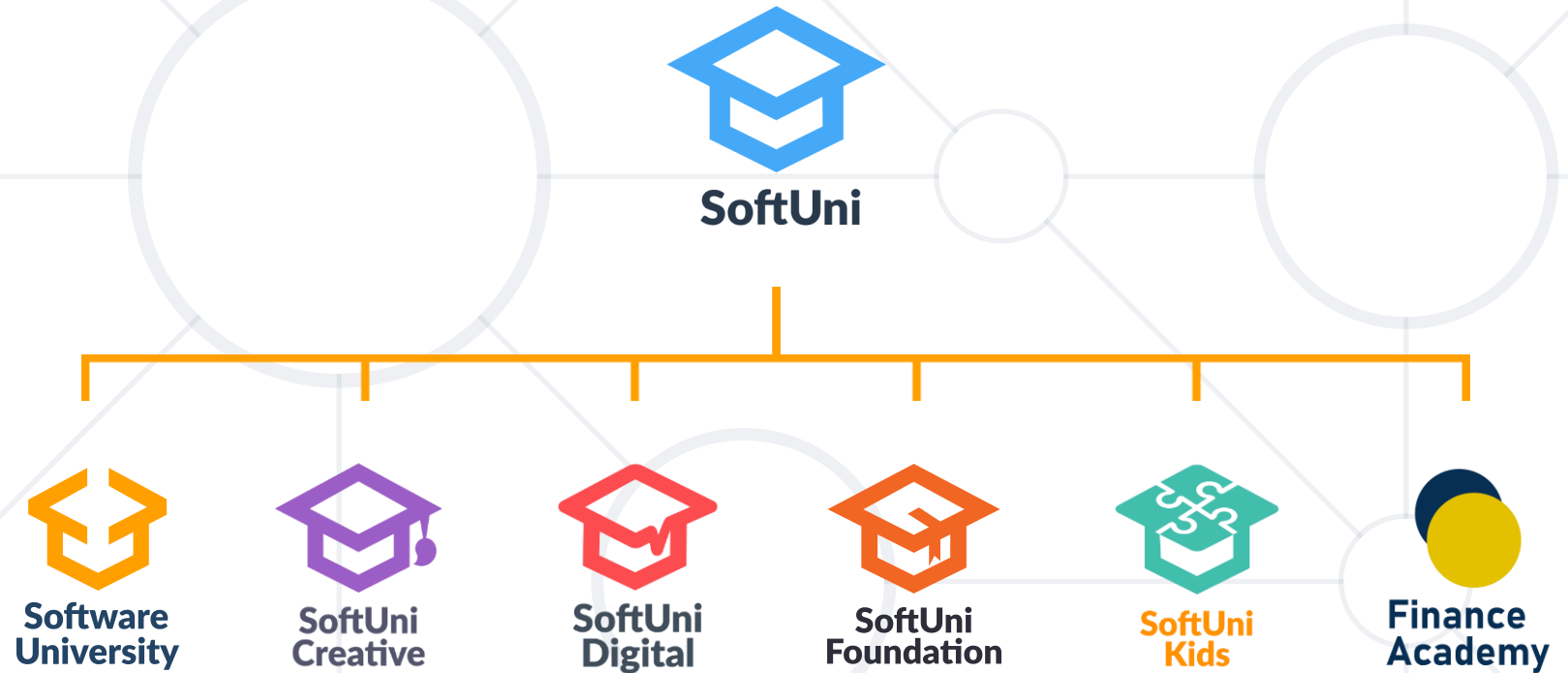
# Solution: From JSON to HTML Table

```
function jsonToHtmlTable(json) {  
  let arr = JSON.parse(json);  
  let outputArr = ["<table>"];  
  outputArr.push(makeKeyRow(arr));  
  arr.forEach((obj) => outputArr.push(makeValueRow(obj)));  
  outputArr.push("</table>");  
  
  function makeKeyRow(arr) { // ToDo }  
  function makeValueRow(obj) { // ToDo };  
  function escapeHtml(value) { // ToDo };  
  
  console.log(outputArr.join('\n'));  
}
```

- Objects
  - Hold **key-value** pairs called **properties**
  - **Methods** are **actions** that can be performed on objects
- Object Context in methods
  - "**this**" keyword
- Composition – combining **complex** objects into **simple** ones
- JSON - **data** interchange **format**



# Questions?



# SoftUni Diamond Partners

**SUPER  
HOSTING  
.BG**



**Coca-Cola HBC  
Bulgaria**

 **Flutter**<sup>TM</sup>  
International

**INDEAVR**  
Serving the high achievers



**AMBITIONED**

 **DRAFT  
KINGS**

 **SOFTWARE  
GROUP**



**BOSCH**

 **Postbank**  
*Решения за твоето утре*

 **PHAR  
VISION**



**SmartIT**

**DXC**  
TECHNOLOGY

**createX**

- This course (slides, examples, demos, exercises, homework, documents, videos and other assets) is **copyrighted content**
- Unauthorized copy, reproduction or use is illegal
- © SoftUni – <https://about.softuni.bg/>
- © Software University – <https://softuni.bg>



- Software University – High-Quality Education, Profession and Job for Software Developers

- [softuni.bg](http://softuni.bg), [about.softuni.bg](http://about.softuni.bg)

- Software University Foundation

- [softuni.foundation](http://softuni.foundation)

- Software University @ Facebook

- [facebook.com/SoftwareUniversity](https://facebook.com/SoftwareUniversity)

- Software University Forums

- [forum.softuni.bg](http://forum.softuni.bg)

