# Exercises: Objects, Inheritance and Prototypes

Problems for exercises and homework for the [“JavaScript Advanced” course @ SoftUni](https://softuni.bg/courses/javascript-advanced). Submit your solutions in the SoftUni judge system at <https://judge.softuni.bg/Contests/301/>.

# Section I

The solutions to the following problems **must** be submitted using the Mocha unit tests strategy. You can select it from the drop-down menu under the input area, near the **[Submit]** button.

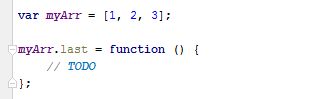
## Array extension

**Extend** the build-in **Array** object with additional functionality. Implement the following functions:

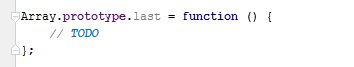
* last() – returns the last element of the array
* skip(n) – returns a new array which includes all original elements, except the first **n** elements
* take(n) – returns a new array containing the first **n** elements from the original array
* sum() – returns a sum of all array elements
* average() – returns the average of all array elements

### Hints

If we have an **instance** of and array, since we know it’s an object, adding new properties to it is pretty straightforward:



This however, only adds our new function to this instance. To add all functions just one time and have them work on **all arrays** is not much more complicated, we just have to attach them to Array’s **prototype** instead:



With such a declaration, we gain access to the context of the calling instance via this. We can then easily access indexes and other existing properties. Don’t forget we don’t want to modify the exiting array, but to create a new one. The last element of the array is the same number as the length of the array. However, since the counting starts at zero, we subtract one to get the last index:



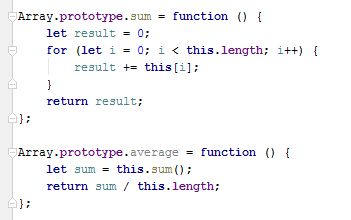
To skip the first **n** elements of an array, we iterate over all elements, starting from index **n** and going through to the end of it. All elements we iterate over this way are added to the result:



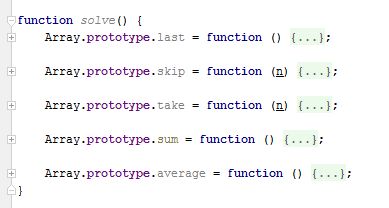
Taking the first n elements is similar – we start from the beginning and iterate to index n, exclusively:



Note these functions do not have any error checking – if **n** is negative or outside the bounds of the array, an exception will be thrown, so take care when using them, or add your own validation. The last two functions require a little bit of arithmetic to be performed:



To test our program in the Judge, we need to wrap it in a function, like this:



We are ready to submit our solution.

## Construction Crew

Write a JS program that **receives** a worker **object** as a **parameter** and modifies its properties. Workers have the following structure:

{ weight: Number,

experience: Number,

bloodAlcoholLevel: Number,

handsShaking: Boolean }

**Weight** is expressed in **kilograms**, **experience** in **years** and **bloodAlcoholLevel** is in **milliliters**. If you receive a worker who’s **handsShaking** property is set to **true** it means he needs to intake some alcohol in order to be able to work correctly. The required amount is 0.1ml per **kilogram** per year of **experience**. The required amount must be **added** to the **existing amount**. Once the alcohol is administered, change the **handsShaking** property to **false**.

Workers whose hands aren’t shaking should **not** be modified in any way. Return them as they were.

### Input

Your function will receive a valid **object** as **parameter**.

### Output

There is no output, **modify** the passed in **object** where necessary.

### Examples

|  |  |
| --- | --- |
| Input | Output |
| { weight: 80,  experience: 1,  bloodAlcoholLevel: 0,  handsShaking: true } | { weight: 80,  experience: 1,  bloodAlcoholLevel: 8,  handsShaking: false } |
| { weight: 120,  experience: 20,  bloodAlcoholLevel: 200,  handsShaking: true } | { weight: 120,  experience: 20,  bloodAlcoholLevel: 440,  handsShaking: false } |
| { weight: 95,  experience: 3,  bloodAlcoholLevel: 0,  handsShaking: false } | { weight: 95,  experience: 3,  bloodAlcoholLevel: 0,  handsShaking: false } |

## Car Factory

Write a JS program that assembles a car by **given requirements** out of **existing** **components**. The client will place an order in the form of an **object describing** the car. You need to **determine** which parts to use to fulfil the client’s order. You have the following parts in storage:

An **engine** has **power** (given in horsepower) and **volume** (given in cubic centimeters). Both of these values are **numbers**. When selecting an engine, pick the **smallest possible** that still meets the requirements.

Small engine: { power: 90, volume: 1800 }

Normal engine: { power: 120, volume: 2400 }

Monster engine: { power: 200, volume: 3500 }

A **carriage** has a **type** and **color**. Both of these values are **strings**. You have two types of carriages in storage and can paint it **any color**.

Hatchback: { type: 'hatchback', color: <as required> }

Coupe: { type: 'coupe', color: <as required> }

The **wheels** will be represented by an **array** of 4 **numbers**, each number represents the **diameter** of the wheel in inches. The size can only be an **odd number**. Round **down** any requirements you receive to the nearest odd number.

### Input

You will receive an **object** as an **argument** to your function. The format will be as follows:

{ model: <model name>,

power: <minimum power>,

color: <color>,

carriage: <carriage type>,

wheelsize: <size> }

### Output

**Return** the resulting car **object** as a result of your function. See the examples for details.

### Examples

|  |  |
| --- | --- |
| Sample input | Output |
| { model: 'VW Golf II',  power: 90,  color: 'blue',  carriage: 'hatchback',  wheelsize: 14 } | { model: 'VW Golf II',  engine: { power: 90,  volume: 1800 },  carriage: { type: 'hatchback',  color: 'blue' },  wheels: [13, 13, 13, 13] } |

|  |  |
| --- | --- |
| Sample input | Output |
| { model: 'Opel Vectra',  power: 110,  color: 'grey',  carriage: 'coupe',  wheelsize: 17 } | { model: 'Opel Vectra',  engine: { power: 120,  volume: 2400 },  carriage: { type: 'coupe',  color: 'grey' },  wheels: [17, 17, 17, 17] } |

## Extensible object

Create an object that can have functionality easily **added** to it, depending on the context. Your object needs to have a unique id property and a single function extend(template), which receives an **object** as argument. The ID should be assigned **automatically** upon instantiation of the object and can **only be read**, but **not changed** later. Upon calling this extend(), your object should assume **all of the properties** of the passed in argument and be able to use them as its own. It’s best if the functions aren’t cloned, but shared between all instances of your object.

### Hints

Lookup online how to extend objects in JavaScript via mixins.

## String extension

**Extend** the build-in String object with additional functionality. Implement the following functions:

* ensureStart(str) – append **str** to the beginning of a string, only if it’s not already present
* ensureEnd(str) – append **str** to the end of a string, only if it’s not already present
* isEmpty() – return true if the string is empty, false otherwise
* truncate(n) – truncates the string to **n** characters by removing words and appends an ellipsis to the end
* format(string, …params) – static method to replace placeholders with parameters

Note strings are **immutable**, so your functions will return new strings as a result instead.

## \*Sorted List

Implement a **collection**, which keeps a list of numbers, sorted in **ascending order**. It must support the following functions:

* add(elemenent) – adds a new element to the collection
* remove(index) – removes the element at position **index**
* get(index) – returns the value of the element at position **index**
* size – number of elements stored in the collection

The **correct order** of the element must be kept **at all times**, regardless of which operation is called. **Removing** and **retrieving** elements **shouldn’t** work if the provided index points **outside the length** of the collection (either throw an error or do nothing). Note the size of the collection is **not** a function. Write your code such that the first function in your solution **returns instance** of your Sorted List.

Input / Output

All function that expect **input** as **parameters** will receive valid data. Any result expected from a function should be **returned** as it’s result. Your **main function** should **return** an **object** instance with the required functionality as it’s result.

# Section II

The solutions to the following problems **must** be submitted using the DOM unit tests strategy. You can select it from the drop-down menu under the input area, near the **[Submit]** button.

## DOM Traversal

Write a program that recursively **traverses** all child nodes of an HTML element and **highlights** a path to the **deepest node**. Your script will be placed inside an HTML page and a **selector** will be passed to it. Starting from a given node and searching downwards, once the node with the deepest nesting is found, add the class highlight to it to change its appearance. Add the same class to **all parent nodes** all the way to the **selector**. If two elements have the same depth, highlight the **first encountered**.

### Input

Your function will be placed inside an HTML document and a **selector** to an HTML element will be passed to it.

### Output

There is no output, your program should **modify** the HTML instead.

### Example



Use the provided HTML skeleton to test your code. Note you will need to download and include jQuery manually, if you want to use it in your local tests. You may find it here: <http://jquery.com/download/>

## \*Bug Tracker

Create a JS program for managing bug reports. It must perform as a self-contained module with exposed functionality. Whenever a new element is added, deleted or changed with a command, the HTML should be updated automatically. A bug report has the following structure:

{ ID: Number,

author: String,

description: String,

reproducible: Boolean,

severity: Number,

status: String }

The **ID** of each report has to be a **unique** number, starting from **zero** and increasing **sequentially**. The module needs to implement the following **functions**:

report(author, description, reproducible, severity) – **create** a new bug report and store it. The **ID** is assigned automatically to the next available number and the **status** defaults to **'Open'**

setStatus(id, newStatus) – change the status of a bug registered in the system to **newStatus** by given **ID**

remove(id) – delete a bug report by given **ID**

sort(method) – change the order in which bug reports are displayed on the webpage. The **method** argument is a string and can be either **'author'**, **'severity'** or **'ID'**. Always sort in ascending order (default behavior for alphabetical sort). The default sorting method is by **'ID'**.

output(selector) – set the HTML element inside which the result is to be displayed to **selector**

Use the following structure for each HTML report:

|  |
| --- |
| HTML |
| <div id="report\_${ID}" class="report">  <div class="body">  <p>${description}</p>  </div>  <div class="title">  <span class="author">Submitted by: ${author}</span>  <span class="status">${status} | ${severity}</span>  </div>  </div> |

Use the provided HTML skeleton to test your solution locally.

### Input

Input will be passed to each applicable function as **parameters** in the correct format.

### Output

Your solution must **expose a module** with all required functions bundled in it (**return** it as a result of your function). The **HTML** should be **modified** as specified.