# Exercises: Objects & Composition - Bonus

Problems for exercises and homework 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/2759/Objects-and-Composition-Exercise>.

## Calorie Object

Write a function that composes an object by given properties. The input comes as an **array of strings**. Every **even index** of the array represents the **name of the food**. Every **odd index** is a **number** that is equal to the **calories in 100 grams of the given product**. Assign each value to its corresponding property and print it on the console.

The **input** comes as an **array of string** **elements**.

The **output** should be printed on the console.

**Examples**

|  |  |
| --- | --- |
| **Input** | **Output** |
| ['Yoghurt', '48', 'Rise', '138', 'Apple', '52'] | { Yoghurt: 48, Rise: 138, Apple: 52 } |
| ['Potato', '93', 'Skyr', '63', 'Cucumber', '18', 'Milk', '42'] | { Potato: 93, Skyr: 63, Cucumber: 18, Milk: 42 } |

## Construction Crew

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

{ weight: Number,

experience: Number,

levelOfHydrated: Number,

dizziness: Boolean }

Weight is expressed in **kilograms**, **experience** in **years** and **levelOfHydrated** is in **milliliters**. If you receive a worker who’s **dizziness** property is set to **true** it means he needs to intake some **water** 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 water is administered, change the **dizziness** property to **false**.

Workers who **do not have dizziness** should **not** be modified in any way. Return them as they were.

### Input

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

### Output

Return the **same object** that was passed in, **modified** as necessary.

### Examples

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

## Car Factory

Write a 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] } |
| { 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] } |

## Rectangle

Write a **function** that creats and returns a rectangle object. The rectangle needs to have a **width** (Number), **height** (Number) and **color** (String) properties, which are set via arguments during creation, and a calcArea() method, that calculates and **returns** the rectangle’s area.

### Input

The function will receive three valid parameters – **width** (Number), **height** (Number) and **color** (String).

### Output

Your function must return an object with all properties and methods as described. The calcArea() method of the object should **return** a number. The first letter in the color must be **upperCase().**

### Examples

|  |  |
| --- | --- |
| Sample Input | Output |
| let rect = rectangle(4, 5, 'red');  console.log(rect.width);  console.log(rect.height);  console.log(rect.color);  console.log(rect.calcArea()); | 4  5  Red  20 |

## Sorted List\*

Create a function that returns a special **object**, which **keeps** a list of numbers, sorted in **ascending order**. It must support the following functionality:

* **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 elements 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.

**Input / Output**

The initial function takes no arguments and must **return** an **object**.

All methods on the object that expect **input** will receive data as **parameters**. Methods that have **validation** will be tested with both **valid and invalid** data. Any result expected from a method should be **returned** as it’s result.

**Examples**

|  |  |
| --- | --- |
| **Sample Input** | **Output** |
| **let list = createSortedList();**  **list.add(5);**  **list.add(6);**  **list.add(7);**  **console.log(list.get(1));**  **list.remove(1);**  **console.log(list.get(1));** | **6**  **7** |

## Heroes

Create a function **returns** an **object** with 2 methods (**mage** and **fighter**). This object should be able to **create** heroes (fighters and mages). Every hero has a **state**.

* Fighters have **name**, **health = 100** and **stamina = 100** and every fighter can fight. When he **fights** his **stamina** **decreases** by **1** and the following message is **printed** on the console:

**`${fighter's name} slashes at the foe!`**

* Mages also have state (**name**, **health = 100** and **mana = 100**). Every mage can **cast** **spells**. When a spell is casted the mage's **mana** **decreases** by **1** and the following message is **printed** on the console:

**`${mage's name} cast ${spell}`**

### Note:

For more information check the examples below.

|  |  |
| --- | --- |
| Input | Output |
| let create = solve();  const scorcher = create.mage("Scorcher");  scorcher.cast("fireball")  scorcher.cast("thunder")  scorcher.cast("light")  const scorcher2 = create.fighter("Scorcher 2");  scorcher2.fight()  console.log(scorcher2.stamina);  console.log(scorcher.mana); | **Scorcher cast fireball**  **Scorcher cast thunder**  **Scorcher cast light**  **Scorcher 2 slashes at the foe!**  **99**  **97** |



## Jan's Notation \*

Write a program that parses a series of instructions written in **postfix notation** and executes them (postfix means the operator is written **after** the operands). You will receive a **series of instructions** – if the instruction is a **number**, **save it**; otherwise, the instruction is an **arithmetic operator** (**+-\*/**) and you must apply it to the most two **most recently saved** numbers. **Discard** these two numbers and in their place, **save the result** of the operation – this number is now eligible to be an **operand** in a subsequent operation. Keep going until all input instructions have been exhausted, or you encounter an **error**.

In the end, if you’re left with a **single saved number**, this is the **result** of the calculation and you must **print** it. If there are more numbers saved, then the user supplied **too many instructions** and you must print "**Error: too many operands!**". If at any point during the calculation you **don’t have** two number saved, the user supplied **too few instructions** and you must print "**Error: not enough operands!**". *See the examples for more details.*

**Input**

You will receive an array with numbers **and** strings – the numbers will be **operands** and must be saved; the strings will be **arithmetic operators** that must be applied to the operands.

**Output**

Print on the **console** on a single line the **final result** of the calculation or an **error message**, as instructed above.

**Constraints**

* The **numbers** (operands) will be integers
* The **strings** (operators) will always be one of **+-\*/**
* The result of each operation will be in range [-253…253-1] (**MAX\_SAFE\_INTEGER** will **never** be exceeded)

**Examples**

|  |  |  |
| --- | --- | --- |
| **Input** | **Output** | **Explanation** |
| [3,  4,  '+'] | 7 | The first instruction is a **number**, therefor we **save** it. The next one is also a **number**, we **save** it too.  The third instruction is a **string**, so it must be an **operator** – we **remove the last two** numbers we saved, and perform the operation: **3+4=7**. The result of this operation is then **saved** where the two operands **used to be**.  We’ve ran out of instructions, so we check the saved values – we only have **one**, so this must be **final result**. We **print** it on the console. |
| [5,  3,  4,  '\*',  '-'] | -7 | We save in order **5**, **3** and **4**. The result of the operation **3\*4** is **12**, which we **save in place** of **3** and **4**.  Currently we have **5** and **12** saved. The result of the operation **5-12** is **-7**, which we **save in place** of **5** and **12**.  We have no more instructions and **only one** value saved, which we **print**. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Input** | **Output** |  | **Input** | **Output** |
| [7,  33,  8,  '-'] | Error: too many operands! | [15,  '/'] | Error: not enough operands! |