Exercises: Objects & Composition

Problems for exercises and homework for the "JavaScript Advanced" course @ SoftUni. Submit your solutions in the SoftUni judge system at https://judge.softuni.bg/Contests/2759/Objects-and-Composition-Exercise.

1. 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 finally print the object.

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 }`

2. 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 whose **`dizziness`** property is set to **`true`** it means he needs to intake some **water** 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 (to the `levelOfHydrated`)**. 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 a **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 }`

3. Car Factory

Write a program that assembles a car by **giving 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 fulfill 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 them **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] }`

4. Heroic Inventory

In the era of heroes, every hero has his items that make him unique. Create a function that creates a **register for the heroes**, with their **names**, **level**, and **items**, if they have such. The register should accept data in a specified format, and return it presented in a specified format.

Input

The **input** comes as an array of strings. Each element holds data for a hero, in the following format:

"**`{heroName} / {heroLevel} / {item1}, {item2}, {item3}...`**"

You must store the data about every hero. The **name** is a **string**, a **level** is a **number** and the items are all **strings.**

Output

The **output** is a **JSON representation** of the data for all the heroes you've stored. The data must be an **array of all the heroes**. Check the examples for more info.

Examples

Input	Output
['Isacc / 25 / Apple, GravityGun', **'Derek / 12 / BarrelVest, DestructionSword',** **'Hes / 1 / Desolator, Sentinel, Antara']**	[{"name":"Isacc","level":25,"items":["Apple","GravityGun"]},{"name":"Derek","level":12,"items":["BarrelVest","DestructionSword"]},{"name":"Hes","level":1,"items":["Desolator","Sentinel","Antara"]}]
['Jake / 1000 / Gauss, HolidayGrenade']	[{"name":"Jake","level":1000,"items":["Gauss","HolidayGrenade"]}]

Hints

- We need an array that will hold our hero data. That is the first thing we create.

```
1   function heroicInventory(input) {
2       let result = [];
```

- Next, we need to loop over the whole input and process it. Let's do that with a simple **for** loop.

```
1   function heroicInventory(input) {
2       let result = [];
3
4       for (const iterator of input) {
5           let [name, level, items] = iterator.split(' / ');
6           level = Number(level);
```

- Every element from the input holds data about a hero, however, the **elements from the data** we need are **separated by some delimiter**, so we just split each string with that **delimiter**.
- Next, we need to take the elements from the **string array**, which is a result of the **string split**, and by destructuring assignment syntax, we assign the array properties. Don't forget to parse the number.

- However, here we remember there is something special about the items, so read the problem definition again, you will notice that there might be a **case** where the hero **has no items**; in that case, using **destructuring** is ok and when there are no items, our property items will be undefined and trying to spit it will throw an error. That is why we need to perform a simple check using the ternary operator.

```
7           items = items ? items.split(', ') : [];
```

- If **there are any items** in the **input**, the **variable** will be set to the **split version of them**. If not, it will just be set to an **empty array**.
- We have now extracted the needed data – we have stored the **input name** in a **variable**, we have parsed the **given level** to a **number**, and we have also **split** the **items** that the **hero holds** by their **delimiter**, which would result in a **string array** of elements. By definition, the **items** are **strings**, so we don't need to process the array we've made anymore.
- Now what is left is to add that data into **an object** and **add** that object to the **array**.

```
4       for (const iterator of input) {
5           let [name, level, items] = iterator.split(' / ');
6           level = Number(level);
7           items = items ? items.split(', ') : [];
8
9           result.push({name, level, items});
10      }
```

- Lastly, we need to turn the array of objects we have made, into a JSON string, which is done by the **`JSON.stringify()`** function

```
12      console.log(JSON.stringify(result));
13  }
```

5. Lowest Prices in Cities

You will be given several towns, with products and their price. You need to find **the lowest price** for **every product** and **the town it is sold at** for that price.

Input

The **input** comes as an array of strings. Each element will hold data about a **town**, **product**, and **its price** at that town. The **town** and **product** will be **strings**, the **price** will be a **number**. The input will come in the following format:

`{townName} | {productName} | {productPrice}`

Output

As **output**, you must print **each product** with its **lowest price** and **the town** at which the product is **sold at that price**. If **two towns share** the **same lowest price**, print the one that was **entered first**.
The output, for every product, should be in the following format:

`{productName} -> {productLowestPrice} ({townName})`

The **order of output** in - **order of entrance**. See the examples for more info.

Examples

Input	Output
['Sample Town | Sample Product | 1000', 'Sample Town | Orange | 2', 'Sample Town | Peach | 1', 'Sofia | Orange | 3', 'Sofia | Peach | 2', 'New York | Sample Product | 1000.1', 'New York | Burger | 10']	Sample Product -> 1000 (Sample Town) Orange -> 2 (Sample Town) Peach -> 1 (Sample Town) Burger -> 10 (New York)

6. Store Catalogue

You have to create a sorted catalog of store products. You will be given the products' names and prices. You need to order them in **alphabetical order**.

Input

The **input** comes as an array of strings. Each element holds info about a product in the following format:

"`{productName} : {productPrice}`"

The **product's name** will be a **string**, which will **always start with a capital letter**, and the **price** will be **a number**. There will be **NO duplicate product input**. The comparison for alphabetical order is **case-insensitive**.

Output

As **output**, you must print all the products in a specified format. They must be ordered **exactly as specified above**. The products must be **divided into groups**, by the **initial of their name**. The **group's initial should be printed**, and after that, the products should be printed with **2 spaces before their names**. For more info check the examples.

Examples

Input	Output
['Appricot : 20.4',	A

Input	Output
['Banana : 2',	B

'Fridge : 1500', 'TV : 1499', 'Deodorant : 10', 'Boiler : 300', 'Apple : 1.25', 'Anti-Bug Spray : 15', 'T-Shirt : 10']	Anti-Bug Spray: 15 Apple: 1.25 Appricot: 20.4 B Boiler: 300 D Deodorant: 10 F Fridge: 1500 T T-Shirt: 10 TV: 1499

'Rubic's Cube : 5', 'Raspberry P : 4999', 'Rolex : 100000', 'Rollon : 10', 'Rali Car : 2000000', 'Pesho : 0.000001', 'Barrel : 10']	Banana: 2 Barrel: 10 P Pesho: 0.000001 R Rali Car: 2000000 Raspberry P: 4999 Rolex: 100000 Rollon: 10 Rubic's Cube: 5

7. Towns to JSON

You're tasked to create and print a JSON from a text table. You will receive input as an array of strings, where each string represents a row of a table, with values on the row encompassed by pipes **"|"** and optionally spaces. The table will consist of exactly 3 columns **"Town"**, **"Latitude"** and **"Longitude"**. The **Latitude** and **Longitude** columns will always contain **valid numbers**. Check the examples to get a better understanding of your task.

Input

The **input** comes as an array of strings – the first string contains the table's headings, each next string is a row from the table.

Output

- The **output** should be an array of objects wrapped in **JSON.stringify()**.
- **Latitude** and **Longitude** must be parsed to **numbers,** and represented till the **second digit after the decimal point**!

Examples

Input	Output
['| Town | Latitude | Longitude |', '| Sofia | 42.696552 | 23.32601 |', '| Beijing | 39.913818 | 116.363625 |']	[{"Town":"Sofia", "Latitude":42.7, "Longitude":23.32 }, {"Town":"Beijing", "Latitude":39.91, "Longitude":116.36 }]
['| Town | Latitude | Longitude |', '| Veliko Turnovo | 43.0757 | 25.6172 |', '| Monatevideo | 34.50 | 56.11 |']	[{"Town":"Veliko Turnovo", "Latitude":43.08, "Longitude":25.62 }, {"Town":"Monatevideo", "Latitude":34.5, "Longitude":56.11 }]

8. Rectangle

Write a **function** that creates 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`

9. 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(element)`** - 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 its result.

Examples

Sample Input	Output

`let list = createSortedList();`
`list.add(5);`
`list.add(6);`
	`6`
	`7`

`list.add(7);`
`console.log(list.get(1));`
`list.remove(1);`
`console.log(list.get(1));`	

10. Heroes

Create a function that **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 a **`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 cast 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`

```
function solve() {
    const canCast = (state) => ({
        cast: (spell) => {
            console.log(`${state.name} cast ${spell}`);
            state.mana--;
        }
    })

    const canFight = (state) => ({

        fight: () => {
            console.log(`${state.name} slashes at the foe!`)
            state.stamina--;
        }

    })

    const fighter = (name) => {
        let state = {
            name,
            health: 100,
            stamina: 100
        }

        return Object.assign(state, canFight(state));
    }

    const mage = (name) => {
        let state = {
            name,
            health: 100,
            mana: 100
        }
        return Object.assign(state, canCast(state));
    }

    return {mage:mage,fighter: fighter};
}
```

11. 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 numbers 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 the range $[-2^{53}...2^{53}-1]$ (**`MAX_SAFE_INTEGER`** will **never** be exceeded)

Examples

Input	Output	Explanation

`[3,`
`4,`
`'+']`
	7	The first instruction is a **number**, therefore 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 operate: **3+4=7**. The result of this operation is then **saved** where the two operands **used to be**.
		We've run out of instructions, so we check the saved values – we only have **one**, so this must be the **final result**. We **print** it on the console.

`[5,`
`3,`
`4,`
`'*',`
`'-']`
	-7	We save in order **5**, **3,** and **4**. The result of 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
[7, 33, 8, '-']	Error: too many operands!

Input	Output
[15, '/']	Error: not enough operands!