Exercises: Objects & Classes

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/2371/Exercise-Objects-Classes]().

1. Heroic Inventory

In the era of heroes, every hero has his own items which make him unique. Create a function which 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 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**, the **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.

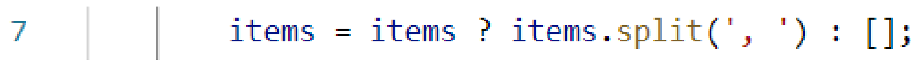
```
function heroicInventory(input) {
    let result = [];
```

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

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

    for (const iterator of input) {
        let [name, level, items] = iterator.split(' / ');
        level = Number(level);
```

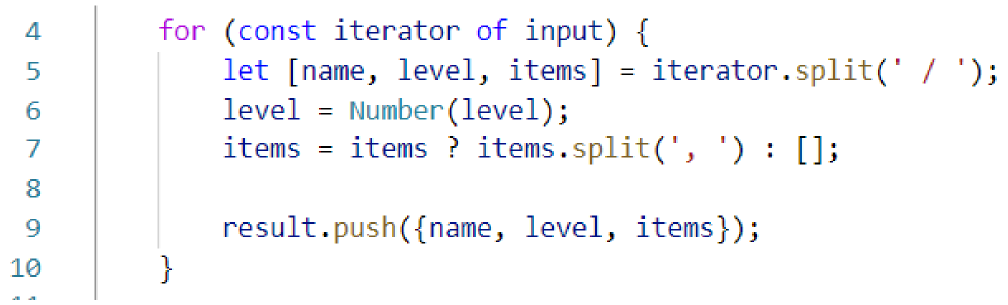
- Every element from the input holds data about a hero, however the **elements from the data** we need are **separated by some delimiter**, so we just split each string with that **delimiter**.
- Next, we need to take the elements from the **string array**, which is a result of the **string split**, 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.



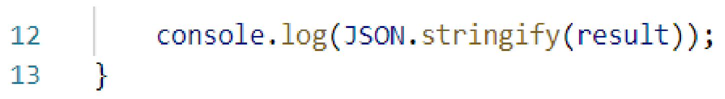
```
        items = items ? items.split(', ') : [];
```

- 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**.



```
    for (const iterator of input) {
        let [name, level, items] = iterator.split(' / ');
        level = Number(level);
        items = items ? items.split(', ') : [];

        result.push({name, level, items});
    }
```

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



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

2. JSON's Table

JSON's Table is a magical table which turns JSON data into an HTML table. You will be given **JSON strings** holding data about employees, including their **name**, **position** and **salary**. You need to **parse that data** into **objects**, and create an **HTML table** which holds the data for each **employee on a different row**, as **columns**.

The **name** and **position** of the employee are **strings**, the **salary** is a **number**.

Input

The **input** comes as array of strings. Each element is a JSON string which represents the data for a certain employee.

Output

The **output** is the HTML code of a table which holds the data exactly as explained above. Check the examples for more info.

Examples

Input	Output

['{"name":"Pesho","position":"Promenliva","salary":100000}',
'{"name":"Teo","position":"Lecturer","salary":1000}',
'{"name":"Georgi","position":"Lecturer","salary":1000}']
	`<table>`
	    `<tr>`
	        `<td>Pesho</td>`
	        `<td>Promenliva</td>`
	        `<td>100000</td>`
	    `</tr>`
	    `<tr>`
	        `<td>Teo</td>`
	        `<td>Lecturer</td>`
	        `<td>1000</td>`
	    `</tr>`
	    `<tr>`
	        `<td>Georgi</td>`
	        `<td>Lecturer</td>`
	        `<td>1000</td>`
	    `</tr>`
	`</table>`

Hints

- You might want to **escape the HTML**. Otherwise you might find yourself victim to vicious JavaScript **code in the input**.

3. Cappy Juice

You will be given different juices, as **strings**. You will also **receive quantity** as a **number**. If you receive a juice, you already have, **you must sum** the **current quantity** of that juice, with the **given one**. When a juice reaches **1000 quantity**, it produces a bottle. You must **store all produced bottles** and you must **print them** at the end.

Note: 1000 quantity of juice is **one bottle**. If you happen to have **more than 1000**, you must make **as much bottles as you can**, and store **what is left** from the juice.

Example: You have 2643 quantity of Orange Juice – this is **2 bottles** of Orange Juice and **643 quantity left**.

Input

The **input** comes as array of strings. Each element holds data about a juice and quantity in the following format:

“**`{juiceName} => {juiceQuantity}`**”

Output

The **output** is the produced bottles. The bottles are to be printed in **order of obtaining the bottles**. Check the second example bellow - even though we receive the Kiwi juice first, we don’t form a bottle of Kiwi juice until the 4th line, at which point we have already create Pear and Watermelon juice bottles, thus the Kiwi bottles appear last in the output.

Examples

Input	Output

['Orange => 2000',
'Peach => 1432',
	Orange => 2
	Peach => 2

'Banana => 450', 'Peach => 600', 'Strawberry => 549']	

['Kiwi => 234',
'Pear => 2345',
'Watermelon => 3456',
'Kiwi => 4567',
'Pear => 5678',
'Watermelon => 6789']
	Pear => 8
	Watermelon => 10
	Kiwi => 4

4. Store Catalogue

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

Input

The **input** comes as 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**. You can safely assume 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',
'Fridge : 1500',
'TV : 1499',
'Deodorant : 10',
'Boiler : 300',
'Apple : 1.25',
'Anti-Bug Spray : 15',
'T-Shirt : 10']
	A
	  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

Input	Output

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

5. Auto-Engineering Company

You are tasked to create a register for a company that produces cars. You need to store **how many cars** have been produced from a **specified model** of a **specified brand**.

Input

The **input** comes as array of strings. Each element holds information in the following format:

“`{carBrand} | {carModel} | {producedCars}`”

The car **brands** and **models** are **strings**, the **produced cars** are **numbers**. If the **car brand** you’ve received **already exists**, just add the **new car model** to it with the **produced cars as its value**. If even the car model exists, just **add** the **given value** to the **current one**.

Output

As **output** you need to print - **for every car brand**, the **car models**, and **number of cars produced** from that model. The output format is:

```
“{carBrand}
  ###{carModel} -> {producedCars}
  ###{carModel2} -> {producedCars}
  ...”
```

The order of printing is the **order in which the brands and models first appear in the input**. The first brand in the input should be the first printed and so on. For each brand, the first model received from that brand, should be the first printed and so on.

Examples

Input	Output
['Audi | Q7 | 1000', 'Audi | Q6 | 100', 'BMW | X5 | 1000', 'BMW | X6 | 100', 'Citroen | C4 | 123', 'Volga | GAZ-24 | 1000000', 'Lada | Niva | 1000000', 'Lada | Jigula | 1000000', 'Citroen | C4 | 22', 'Citroen | C5 | 10']	Audi ###Q7 -> 1000 ###Q6 -> 100 BMW ###X5 -> 1000 ###X6 -> 100 Citroen ###C4 -> 145 ###C5 -> 10 Volga ###GAZ-24 -> 1000000 Lada ###Niva -> 1000000 ###Jigula -> 1000000

6. System Components

You will be given a register of systems with components and subcomponents. You need to build an ordered database of all the elements that have been given to you.

The elements are registered in a very simple way. When you have processed all of the input data, you must print them in a specific order. For every System you must print its components in a specified order, and for every Component, you must print its Subcomponents in a specified order.

The **Systems** you’ve stored must be ordered by **amount of components**, in **descending order**, as **first criteria**, and by **alphabetical order** as **second criteria**. The **Components** must be ordered by **amount of Subcomponents**, in **descending order**.

Input

The **input** comes as array of strings. Each element holds **data** about a **system**, a **component** in that **system**, and a **subcomponent** in that **component**. If the given **system already exists**, you should just **add the new component** to it. If even the **component exists**, you should just **add** the **new subcomponent** to it. The **subcomponents** will **always be unique**. The input format is:

"`{systemName} | {componentName} | {subcomponentName}`"

All of the elements are strings, and can contain **any ASCII character**. The **string comparison** for the alphabetical order is **case-insensitive**.

Output

As **output** you need to print all of the elements, ordered exactly in the way specified above. The format is:

```
"{systemName}
 |||{componentName}
 |||{component2Name}
 ||||||{subcomponentName}
 ||||||{subcomponent2Name}
 {system2Name}
 ..."
```

Examples

Input	Output
['SULS | Main Site | Home Page', 'SULS | Main Site | Login Page', 'SULS | Main Site | Register Page', 'SULS | Judge Site | Login Page', 'SULS | Judge Site | Submittion Page', 'Lambda | CoreA | A23', 'SULS | Digital Site | Login Page', 'Lambda | CoreB | B24', 'Lambda | CoreA | A24', 'Lambda | CoreA | A25', 'Lambda | CoreC | C4', 'Indice | Session | Default Storage', 'Indice | Session | Default Security']	Lambda |||CoreA ||||||A23 ||||||A24 ||||||A25 |||CoreB ||||||B24 |||CoreC ||||||C4 SULS |||Main Site ||||||Home Page ||||||Login Page ||||||Register Page |||Judge Site ||||||Login Page ||||||Submittion Page |||Digital Site ||||||Login Page Indice |||Session ||||||Default Storage ||||||Default Security

Hints

- Creating a sorting function with two criteria might seem a bit daunting at first, but it can be simplified to the following:
    - If elements **a** and **b** are different based on the **first criteria**, then that result is the result of the sorting function, checking the second criteria is not required.
    - If elements **a** and **b** are **equal** based on the **first criteria**, then the result of comparing **a** and **b** on the **second criteria** is the result of the sorting.

7. Data Class

Write a **class** that holds data about an HTTP **Request**. It has the following properties:

- **`method`** (String)
- **`uri`** (String)
- **`version`** (String)
- **`message`** (String)
- **`response`** (String)
- **`fulfilled`** (Boolean)

The first four properties (**`method, uri, version, message`**) are set trough the **`constructor`**, in the listed order. The **`response`** property is initialized to **`undefined`** and the **`fulfilled`** property is initially set to **`false`**.

Constraints

- The constructor of your class will receive **valid parameters**.
- Submit the class definition as is, **without** wrapping it in any function.

Examples

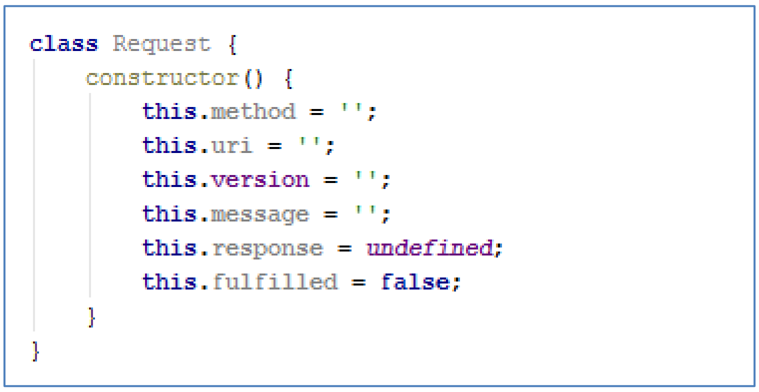
Sample Input	Resulting object
`let myData = new Request('GET', 'http://google.com', 'HTTP/1.1', '')` `console.log(myData);`	`Request {` `method: 'GET',` `uri: 'http://google.com',` `version: 'HTTP/1.1',` `message: '',` `response: undefined,` `fulfilled: false` `}`

Hints

Using ES6 syntax, a class can be defined similar to a function, using the **`class`** keyword:

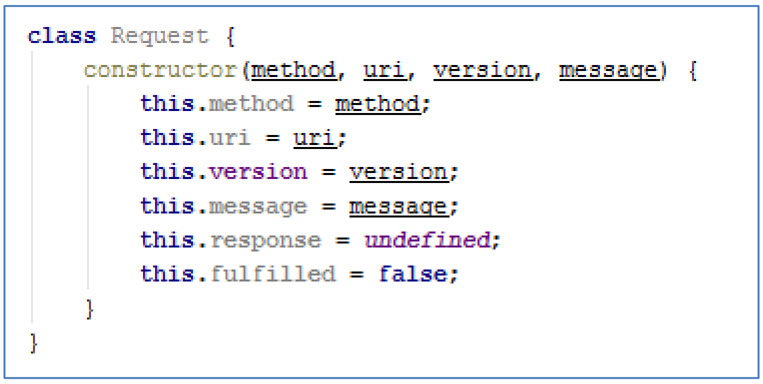
```
class Request {

}
```

At this point, the **class** can already **be instantiated**, but it won't hold anything useful, since it doesn't have a constructor. A **constructor** is a function that **initializes** the object's **context** and attaches **values** to it. It is defined with the keyword **`constructor`** inside the body of the class definition and it follows the syntax of regular JS functions - it can take **arguments** and execute **logic**. Any variables we want to be attached to the **instance** must be prefixed with the **`this`** identifier:



```
class Request {
    constructor() {
        this.method = '';
        this.uri = '';
        this.version = '';
        this.message = '';
        this.response = undefined;
        this.fulfilled = false;
    }
}
```

The description mentions some of the properties need to be set via the constructor - this means the constructor must receive them as parameters. We modify it to take four named parameters that we then assign to the local variables:



```
class Request {
    constructor(method, uri, version, message) {
        this.method = method;
        this.uri = uri;
        this.version = version;
        this.message = message;
        this.response = undefined;
        this.fulfilled = false;
    }
}
```

Note the input parameters have the same names as the instance variables - this isn't necessary, but it's easier to read. There will be no name collision, because the **`this`** identifier tells the interpreter to look for a variable in a different context, so **`this.method`** is not the same as **`method`**.

Our class is complete and can be submitted in Judge.

8. Tickets

Write a program that manages a database of tickets. A ticket has a **`destination`,** a **`price`** and a **`status`**. Your program will receive **two arguments** - the first is an **array of strings** for ticket descriptions and the second is a **string**, representing a **sorting criterion**. The ticket descriptions have the following format:

`<destinationName>|<price>|<status>`

Store each ticket and at the end of execution **return** a sorted summary of all tickets, sorted by either **`destination`**, **`price`** or **`status`**, depending on the **second parameter** that your program received. Always sort in ascending order (default behavior for **alphabetical** sort). If two tickets compare the same, use order of appearance. See the examples for more information.

Input

Your program will receive two parameters - an **array of strings** and a **single string**.

Output

Return a **sorted array** of all the tickets that where registered.

Examples

Sample Input	Output Array

['Philadelphia|94.20|available',
 'New York City|95.99|available',
 'New York City|95.99|sold',
 'Boston|126.20|departed'],
'destination'
	[Ticket { destination: 'Boston',
	    price: 126.20,
	    status: 'departed' },
	  Ticket { destination: 'New York City',
	    price: 95.99,
	    status: 'available' },
	  Ticket { destination: 'New York City',
	    price: 95.99,
	    status: 'sold' },
	  Ticket { destination: 'Philadelphia',
	    price: 94.20,
	    status: 'available' }]

['Philadelphia|94.20|available',
 'New York City|95.99|available',
 'New York City|95.99|sold',
 'Boston|126.20|departed'],
'status'
	[Ticket { destination: 'Philadelphia',
	    price: 94.20,
	    status: 'available' },
	  Ticket { destination: 'New York City',
	    price: 95.99,
	    status: 'available' },
	  Ticket { destination: 'Boston',
	    price: 126.20,
	    status: 'departed' },
	  Ticket { destination: 'New York City',
	    price: 95.99,
	    status: 'sold' }]

9. Sorted List

Implement a **class**, 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

All function that expect **input** will receive data as **parameters**. Functions that have **validation** will be tested with both **valid and invalid** data. Any result expected from a function should be **returned** as it's result.
Your **add** and **remove functions** should **return** an **class instance** with the required functionality as it's result.

Submit the class definition as is, **without** wrapping it in any function.

Examples

Sample Input	Output

`let list = new List();`
`list.add(5);`
`list.add(6);`
`list.add(7);`
`console.log(list.get(1));`
`list.remove(1);`
`console.log(list.get(1));`
	6
	7

10. Length Limit

Create a class **`Stringer`**, which holds **single string** and a **length** property. The class should be initialized with a **string**, and an **initial length.** The class should always keep the **initial state** of its **given string**.

Name the two properties **`innerString`** and **`innerLength`**.

There should also be functionality for increasing and decreasing the initial **length** property.
Implement function **`increase(length)`** and **`decrease(length)`**, which manipulate the length property with the **given value**.

The length property is **a numeric value** and should not fall below **0**. It should not throw any errors, but if an attempt to decrease it below 0 is done, it should be automatically set to **0**.

You should also implement functionality for **`toString()`** function, which returns the string, the object was initialized with. If the length of the string is greater than the **length property**, the string should be cut to from right to left, so that it has the **same length** as the **length property**, and you should add **3 dots** after it, if such **truncation** was **done**.

If the length property is **0**, just return **3 dots.**

Examples

lengthLimit.js

`let test = new Stringer("Test", 5);`
`console.log(test.toString()); // Test`
 	
`test.decrease(3);`
`console.log(test.toString()); // Te...`

```
test.decrease(5);
console.log(test.toString()); // ...

test.increase(4);
console.log(test.toString()); // Test
```

Hints

Store the initial string in a property, and do not change it. Upon calling the `toString()` function, truncate it to the **desired value** and return it.

Submit your solution as a class representation only! No need for IIFEs or wrapping of classes.