Lab: Object Composition

1. 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}` ``

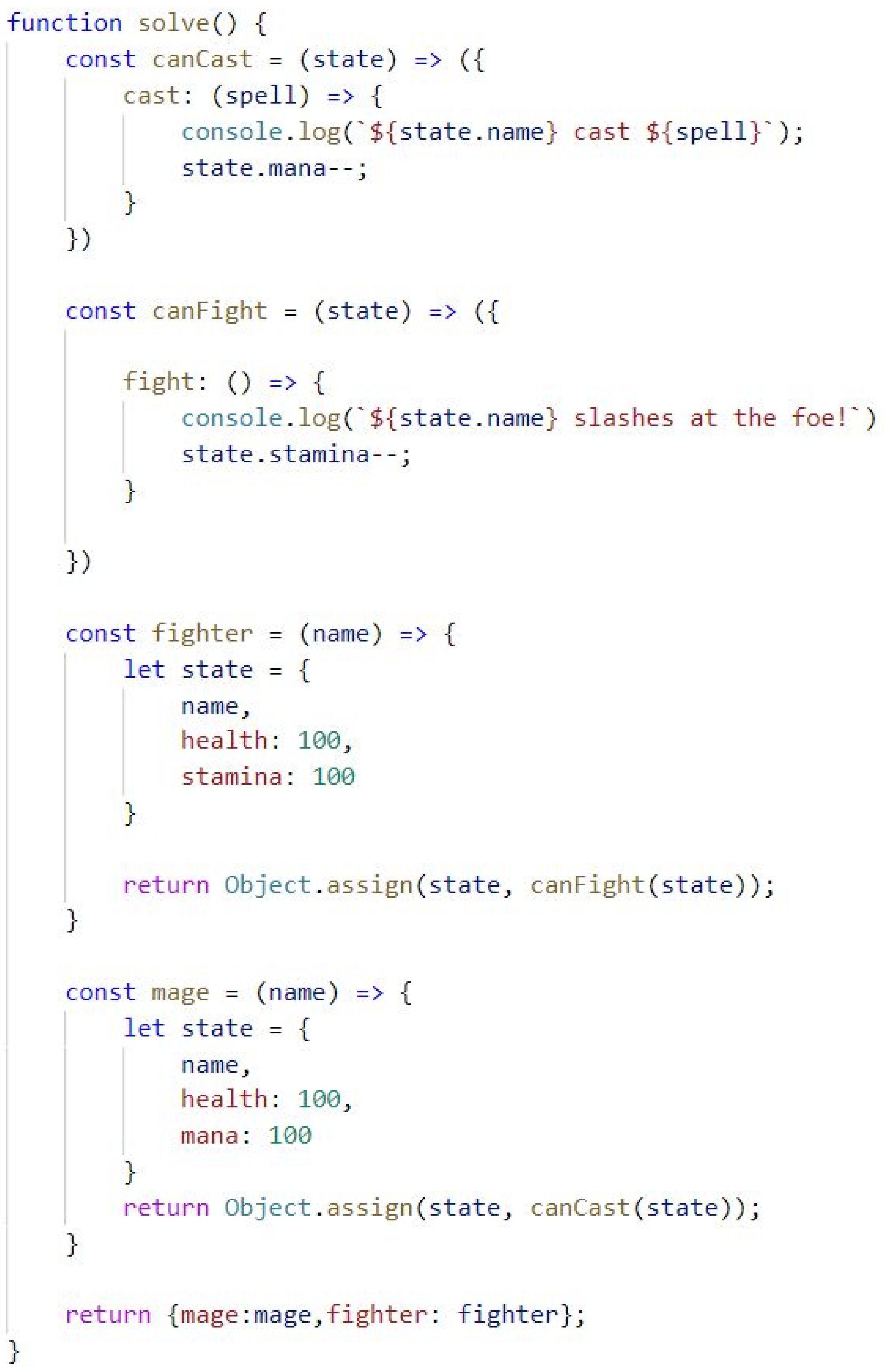
Note:

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`

Hints:



```
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};
}
```

2. Order Rectangles

You will be passed a few pairs of **widths** and **heights** of rectangles, create **objects** to represent the rectangles. The objects should additionally have two functions **`area`** - that returns the area of the rectangle and **`compareTo`** - that compares the current rectangle with another and produces a number signifying if the current rectangle is **smaller** (negative number), **equal** (0) or **larger** (positive number) than the other rectangle.

Input

The input will come as an **array of arrays** - every nested array will contain exactly 2 numbers the **`width`** and the **`height`** of the rectangle.

Output

The output must consist of an array of **rectangles** (objects) sorted by their **`area`** in **descending** order as a **first** criteria and by their **`width`** in **descending** order as a **second** criteria.

Examples

Input	Output

`[[10,5],[5,12]]`	`[{width:5, height:12, area:function(), compareTo:function(other)},`
	 `{width:10, height:5, area:funciton(),compareTo:function(other)}]`

`[[10,5], [3,20], [5,12]]`	`[{width:5, height:12, area:function(), compareTo:function(other)},`
	 `{width:3, height:20, area:funciton(),compareTo:function(other)},`
	 `{width:10, height:5, area:funciton(),compareTo:function(other)}]]`

3. List Processor

Using a closure, create an inner object to process list commands. The commands supported should be the following:

- **add <string>** - adds the following string in an inner collection.
- **remove <string>** - removes all occurrences of the supplied **<string>** from the inner collection.
- **print** - prints all elements of the inner collection joined by **","**.

Input

The **input** will come as an **array of strings** - each string represents a **command** to be executed from the command execution engine.

Output

For every print command - you should print on the console the inner collection joined by **","**

Examples

Input	Output
`['add hello', 'add again', 'remove hello', 'add again', 'print']`	`again,again`
`['add pesho', 'add george', 'add peter', 'remove peter','print']`	`pesho,george`

4. Object Factory

Write a function that can **compose objects**. You will **receive** a **string** and your goal is to create a **new object** with all the **unique** properties you were **given**. For more information check the examples below.

Input

The **input** will come as a **string**, which represents an array of objects.

Output

You should print the **newly created object**.

Examples

Input	Output
`` `[{"canMove": true},{"canMove":true, "doors": 4},{"capacity": 5}]` ``	`{ canMove: true, doors: 4, capacity: 5 }`
`` `[{"canFly": true},{"canMove":true, "doors": 4},{"capacity": 255},{"canFly":true, "canLand": true}]` ``	`{ canFly: true, canMove: true, doors: 4, capacity: 255, canLand: true }`

5. Cars

Write a closure that can create and modify objects. All created objects should be **kept** and be accessible by **name**. You should support the following functionality:

- **`create <name>`** - creates an object with the supplied **`<name>`**
- **`create <name> inherits <parentName>`** - creates an object with the given **`<name>`**, that inherits from the parent object with the **`<parentName>`**
- **`set <name> <key> <value>`** - sets the property with key equal to **`<key>`** to **`<value>`** in the object with the supplied **`<name>`**.
- **`print <name>`** - prints the object with the supplied **`<name>`** in the format **`"<key1>:<value1>,<key2>:<value2>…"`** - the printing should also print all **inherited properties** from parent objects. Inherited properties should come after own properties.

Input

The **input** will come as an **array of strings** - each string represents a **command** to be executed from your closure.

Output

For every **`print`** command - you should print on the console all properties of the object in the above mentioned format.

Constraints

- **All commands will always be valid, there will be no nonexistent or incorrect input.**

Examples

Input	Output

`['create c1',`
`'create c2 inherit c1',`
`'set c1 color red',`
`'set c2 model new',`
`'print c1',`
`'print c2']`
	`color:red`
	`model:new, color:red`

6. Sum

Create a function which returns an object that can modify the DOM. The returned object should support the following functionality:

- **`init(selector1, selector2, resultSelector)`** - initializes the object to work with the elements corresponding to the supplied selectors.
- **`add()`** - **adds** the numerical value of the element corresponding to **`selector1`** to the numerical value of the element corresponding to **`selector2`** and then writes the result in the element corresponding to **resultSelector**
- **`subtract()`** - **subtracts** the numerical value of the element corresponding to **`selector2`** from the numerical value of the element corresponding to **`selector1`** and then writes the result in the element corresponding to **`resultSelector`**

Input

There will be no input your function must only provide an object.

Output

Your function should return an object that meets the specified requirements.

Constraints

- **All commands will always be valid, there will be no nonexistent or incorrect input.**
- **All selectors will point to single textbox elements.**

HTML

You are given the following HTML for testing purposes:

sum.html

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
</head>
<body>
<input type="text" id="num1" />
<input type="text" id="num2" />
<input type="text" id="result" readonly />
<br>
<button id="sumButton">
    Sum</button>
<button id="subtractButton">
    Subtract</button>
</body>
</html>
```