Exercise: Advanced Functions

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/2765/Advanced-Functions-Exercise

1. Sort Array

Write a function that **sorts an array** with **numeric** values in **ascending** or **descending** order, depending on an **argument** that is passed to it.

You will receive a **numeric array** and a **string** as arguments to the function in your code.

- If the second argument is **asc**, the array should be sorted in **ascending order** (smallest values first).
- If it is **desc**, the array should be sorted in **descending order** (largest first).

Input

You will receive a **numeric array** and a **string** as input parameters.

Output

The output should be the **sorted array**.

Examples

Input	Output
`[14, 7, 17, 6, 8], 'asc'`	`[6, 7, 8, 14, 17]`
`[14, 7, 17, 6, 8], 'desc'`	`[17, 14, 8, 7, 6]`

2. Argument Info

Write a function that displays **information** about the **arguments** which are passed to it (**type** and **value**) and a **summary** about the number of each type in the following format:

`` `{argument type}: {argument value}` ``

Print **each** argument description on a **new line**. At the end print a **tally** with counts for each type in **descending order**, each on a **new line** in the following format:

`` `{type} = {count}` ``

If two types have the **same count**, use **order of appearance**.

Do **NOT** print anything for types that do not appear in the list of arguments.

Input

You will receive a series of arguments **passed** to your function.

Output

Print on the console the **type** and **value** of each argument passed into your function.

Example

Input
`'cat', 42, function () { console.log('Hello world!'); }`

Output

`string: cat`
`number: 42`
`function: function () { console.log('Hello world!'); }`
`string = 1`
`number = 1`
`function = 1`

3. Fibonacci

Write a JS function that when called, returns the next Fibonacci number, starting at 0, 1. Use a **closure** to keep the current number.

Input

There will be no input.

Output

The **output** must be a Fibonacci number and must be **returned** from the function.

Examples

Sample execution

```
let fib = getFibonator();
console.log(fib()); // 1
console.log(fib()); // 1
console.log(fib()); // 2
console.log(fib()); // 3
console.log(fib()); // 5
console.log(fib()); // 8
console.log(fib()); // 13
```

4. Breakfast Robot

Your task is to write the management software for a breakfast chef robot - it needs to **take orders**, keep track of available **ingredients** and output an **error** if something's wrong. The cooking instructions have already been installed, so your module needs to **plug into** the system and only take care of **orders** and **ingredients**. And since this is the future and food is printed with nano-particle beams, all ingredients are microelements - **protein**, **carbohydrate**, **fat,** and **flavours**. The library of recipes includes the following meals:

- `apple` - made with **1 carbohydrate** and **2 flavour**
- `lemonade` - made with **10 carbohydrate** and **20 flavour**

- **burger** - made with **5 carbohydrate**, **7 fat** and **3 flavour**
- **eggs** - made with **5 protein**, **1 fat** and **1 flavour**
- **turkey** - made with **10 protein**, **10 carbohydrate**, **10 fat** and **10 flavour**

The robot receives instructions either to **restock** the supply, **cook** a meal, or **report** statistics. The input consists of one of the following commands:

- **restock <microelement> <quantity>** - increases the stored quantity of the given microelement
- **prepare <recipe> <quantity>** - uses the available ingredients to prepare the given meal
- **report** - returns information about the stored microelements, in the order described below, including zero elements

The robot is equipped with quantum field storage, so it can hold an **unlimited quantity** of ingredients, but there is no guarantee there will be enough available to prepare a recipe, in which case an **error message** should be returned. Their availability is checked in the **order** in which they **appear** in the recipe, so the error should reflect the first requirement that was not met.

Submit a **closure** that returns the management function. The management function takes one parameter.

Input

Instructions are passed as a **string argument** to your management function. It will be called **several times** per session, so the internal state must be **preserved** throughout the entire session.

Output

The **return** value of each operation is one of the following strings:

- **Success** - when restocking or completing cooking without errors
- **Error: not enough <ingredient> in stock** - when the robot couldn't muster enough microelements
- **protein={qty} carbohydrate={qty} fat={qty} flavour={qty}** - when a report is requested, in a single string

Constraints

- Recipes and ingredients in commands will always have valid names.

Examples

Execution

```let manager = solution ();```
```console.log (manager ("restock flavour 50")); // Success```
```console.log (manager ("prepare lemonade 4")); // Error: not enough carbohydrate in stock```

Input	Output
restock flavour 50 prepare lemonade 4 restock carbohydrate 10	Success Error: not enough carbohydrate in stock Success

restock flavour 10	Success
prepare apple 1	Success
restock fat 10	Success
prepare burger 1	Success
report	protein=0 carbohydrate=4 fat=3 flavour=55

Input	Output
prepare turkey 1	Error: not enough protein in stock
restock protein 10	Success
prepare turkey 1	Error: not enough carbohydrate in stock
restock carbohydrate 10	Success
prepare turkey 1	Error: not enough fat in stock
restock fat 10	Success
prepare turkey 1	Error: not enough flavour in stock
restock flavour 10	Success
prepare turkey 1	Success
report	protein=0 carbohydrate=0 fat=0 flavour=0

5. Functional Sum

Write a function that **adds** a number passed to it to an **internal sum** and returns **itself** with its internal sum set to the **new value**, so it can be **chained functionally**.

Hint: Overwrite **toString()** of the function.

```
console.log(add(4)(3).toString());
```

Input

Your function needs to take one **numeric argument**.

Output

Your function needs to **return** itself with an updated context.

Example

Input	Output
add(1)	1
add(1)(6)(-3)	4

6. Monkey Patcher *

Your employer placed you in charge of an old forum management project. The client requests new functionality, but the legacy code has high coupling, so you don't want to change anything, for fear of breaking everything else. You know which values need to be accessed and modified, so it's time to monkey patch!

Write a program to extend a forum post record with voting functionality. It needs to have the options to **upvote**, **downvote,** and tally the **total score** (positive minus negative votes). Furthermore, to prevent abuse, if a post has more than 50 **total votes**, the numbers must be obfuscated – the stored values remain the same, but the **reported** amounts of upvotes and downvotes have a number **added** to them. This number is 25% of the **greater number** of votes (positive or negative), rounded up. The actual numbers should **not be modified**, just the reported amounts.

Every post also has a **rating**, depending on its score. If **positive** votes are the overwhelming majority (>66%), the rating is **hot**. If there is no majority, but the balance is non-negative and the **sum** of both votes is more than 100, its rating is **controversial**. If the balance is negative, the rating becomes **unpopular**. If the post has less than 10 **total** votes, or no other rating is met, its rating is **new** regardless of balance. These calculations are performed on the actual numbers.

Your function will be invoked with the **call(object, arguments)**, so treat it as though it is internal for the object. A forum post, to which the function will be attached, has the following structure:

<table>
<tr><td align="center">JavaScript</td></tr>
</table>

```
{
  id: <id>,
  author: <author name>,
  content: <text>,
  upvotes: <number>,
  downvotes: <number>
}
```

The arguments will be one of the following strings:

- **upvote** – increase the positive votes by one
- **downvote** – increase the negative votes by one
- **score** – report positive and negative votes, balance and rating in an array; obfuscation rules apply

Input

Input will be passed as arguments to your function through a **call()** invocation.

Output

Output from the report command should be **returned** as a result of the function in the form of an **array** of three **numbers** and a **string**, as described above.

Examples

<table>
<tr><td align="center">Sample execution</td></tr>
</table>

```
let post = {
    id: '3',
    author: 'emil',
```

SoftUni

```
    content: 'wazaaaaa',
    upvotes: 100,
    downvotes: 100
};
solution.call(post, 'upvote');
solution.call(post, 'downvote');
let score = solution.call(post, 'score'); // [127, 127, 0, 'controversial']
solution.call(post, 'downvote');          // (executed 50 times)
score = solution.call(post, 'score');     // [139, 189, -50, 'unpopular']
```

Explanation

The post begins at 100/100, we add one upvote and one downvote, bringing it to 101/101. The reported score is inflated by 25% of the greater value, rounded up (26). The balance is 0, and at least one of the numbers is greater than 100, so we return an array with the rating '**controversial**'.

We downvote 50 times, bringing the score to 101/151, the reported values are inflated by 151*0.25=38 (rounded up), and since the balance is negative with return an array with rating '**unpopular**'.

DOM-Related Problems

The following problems must be solved using DOM manipulation techniques.

Environment Specifics

Please, be aware that every JS environment may **behave differently** when executing code. Certain things that work in the browser are not supported in **Node.js**, which is the environment used by **Judge**.

The following actions are **NOT** supported:

- `.forEach()` with **NodeList** (returned by `querySelector()` and `querySelectorAll()`)
- `.forEach()` with **HTMLCollection** (returned by `getElementsByClassName()` and `element.children`)
- Using the **spread-operator** (`...`) to convert a **NodeList** into an array
- `append()` in Judge (use only `appendChild()`)
- `prepend()`
- `replaceWith()`
- `replaceAll()`
- `closest()`
- `replaceChildren()`
- Always turn the collection into a **JS array** (forEach, forOf, et.)

If you want to perform these operations, you may use `Array.from()` to first convert the collection into an array.

7. Simple Calculator

Create a function **calculator** 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 `selector1` from the numerical value of the element corresponding to `selector2` 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.**
- **Use the given skeleton to solve this problem.**

Sample execution
`const calculate = calculator ();` `calculate.init ('#num1', '#num2', '#result');`

8. Next Article

Write a JS program that sequentially **displays articles** on a web page when the user **clicks** a button. You will receive an **array of strings** that will initialize the program. You need to return a function that keeps the initial array in its closure and every time it's called, it takes the first element from the array and displays it on the web page, inside "**article**", in div with ID "**content**". If there are no more elements left, your function should do nothing.

Your function will be called automatically, there is **no need** to attach any event listeners.

Input

You will receive an **array** of strings.

Output

Return a **function** that sequentially displays the array elements on the web page.

Examples
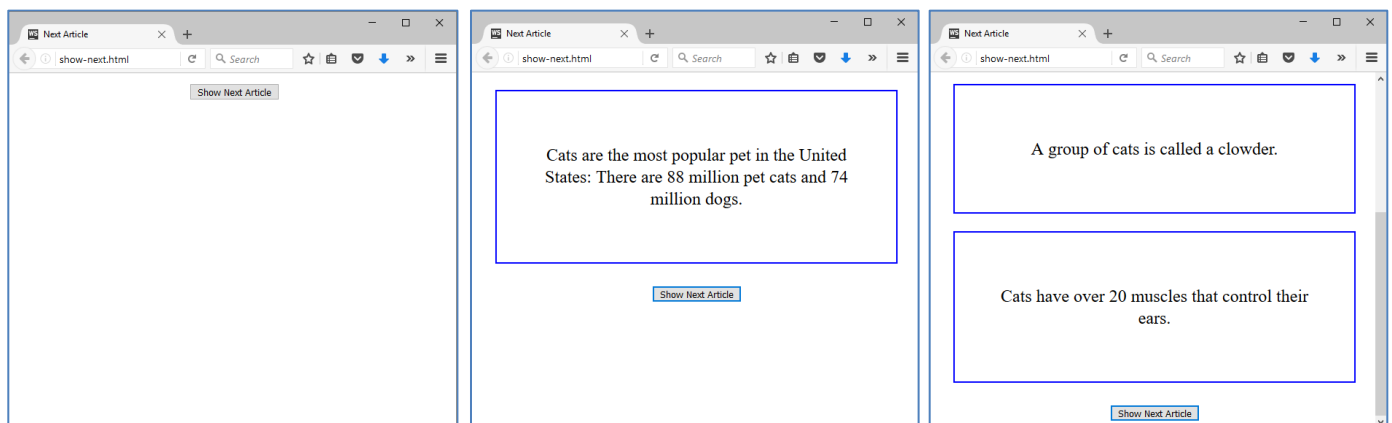
9. Task Manager

Use the **index.html** and **app.js** files to solve this problem. You have **NO permission** to directly change the given HTML code (`index.html` file).

Your task

Write the missing JavaScript code to make the **Task Manager Functionality** works as follows:

When the [**Add**] button is clicked, first you need to validate the inputs. If any of the input fields are empty, the function doesn't make anything.

After validating the input fields, you need to add the new task (**article**) in the "**Open**" section.

The HTML structure looks like this:

```
<section>
    <div>
        <h1 class="orange">Open</h1>
    </div>
    <div>
        <article>
            <h3>JS Advanced Exam</h3>
            <p>Description: To organize the Exam</p>
            <p>Due Date: 2020.04.14</p>
            <div class="flex">
                <button class="green">Start</button>
                <button class="red">Delete</button>
            </div>
        </article>
    </div>
</section>
```

The **article** should have two buttons "**Start**" and "**Delete**". Be careful to set the classes for the buttons and the parent-div.

When the [**Start**] button is clicked, you need to **move** the Task in the section "**In Progress**". Be careful with the buttons! The HTML structure looks like this:

```
<section>
    <div>
        <h1 class="yellow">In Progress</h1>
    </div>
    <div>
        <article>
            <h3>JS Exam preparation</h3>
            <p>Description: To make tasks for JS Advanced Exam</p>
            <p>Due Date: 2020.03.15</p>
            <div class="flex">
                <button class="red">Delete</button>
                <button class="orange">Finish</button>
            </div>
        </article>
    </div>
</section>
```

When the [Delete] button is clicked, the Task (whole article) should be **removed** from the HTML.

After clicking the [Finish] button, the Task will be completed, and you should **move** the **article** to the section "**Complete**". The buttons with their parent div-element should be **removed**.
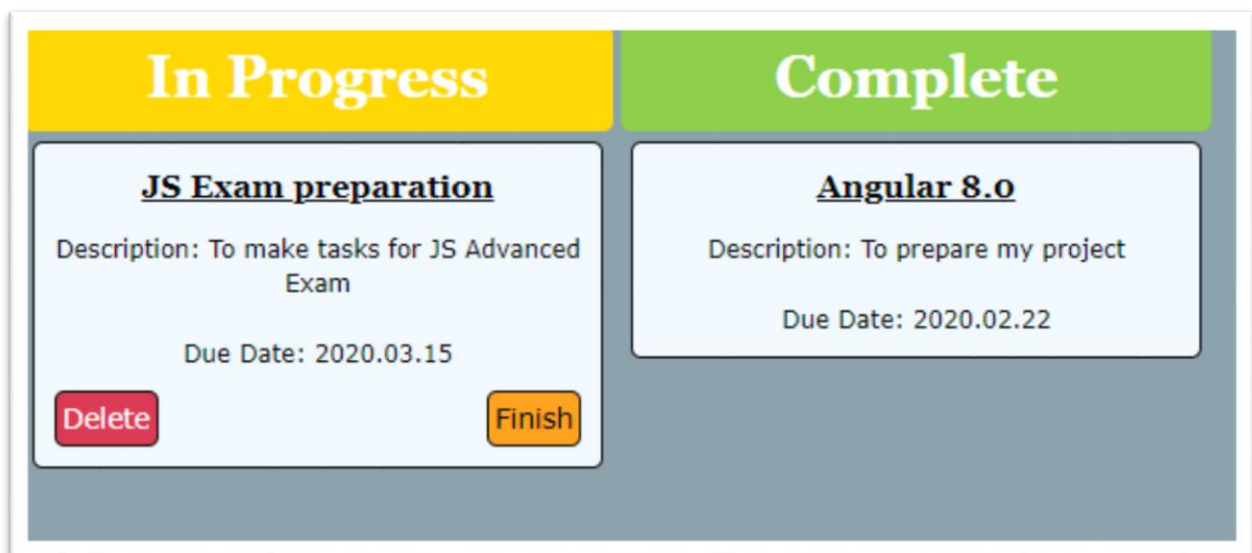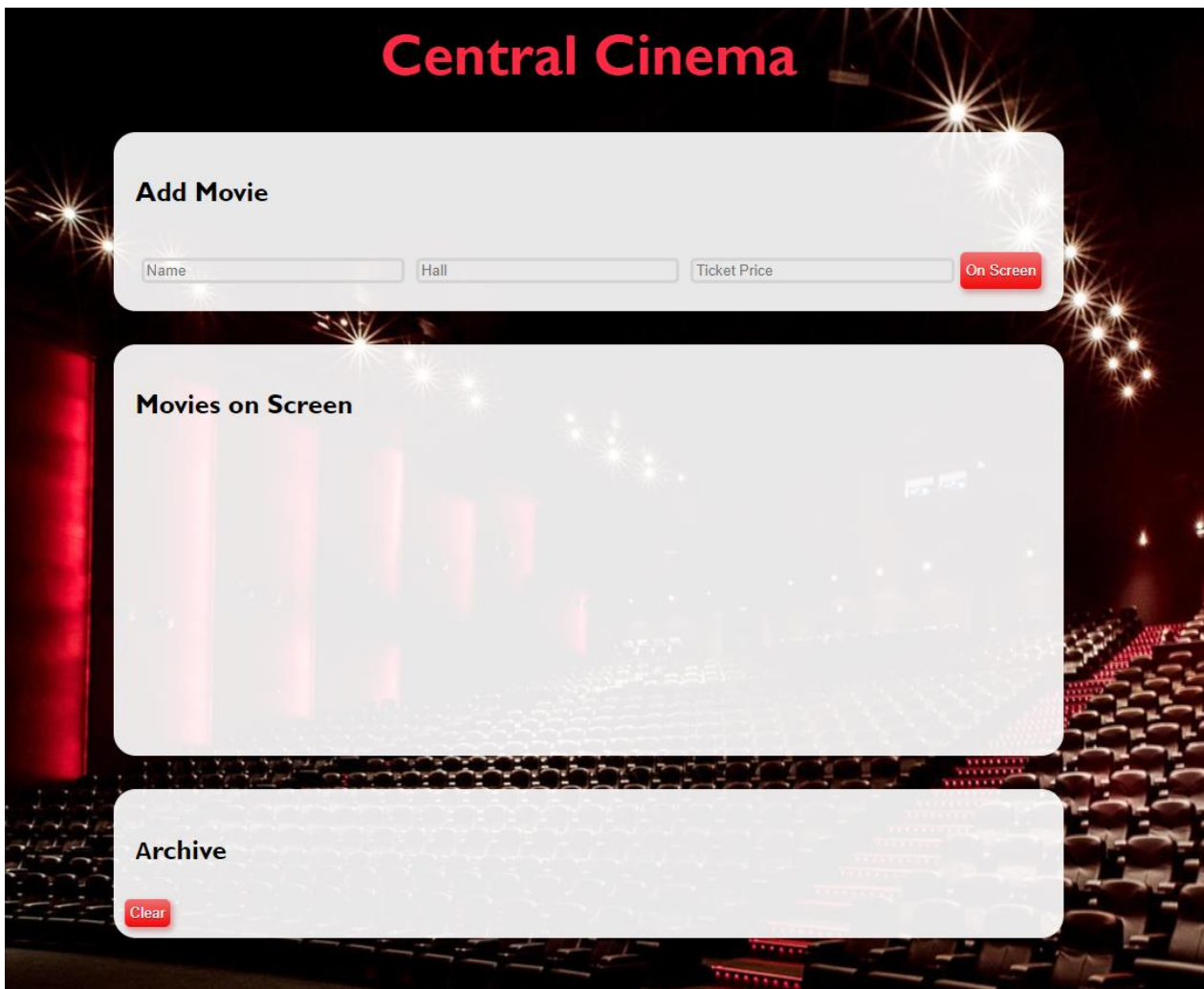

```
<section>
    <div>
        <h1 class="green">Complete</h1>
    </div>
    <div>
        <article>
            <h3>Angular 8.0</h3>
            <p>Description: To prepare my project</p>
            <p>Due Date: 2020.02.22</p>
        </article>
    </div>
</section>
```

10. Central Cinema *

Use the given skeleton to solve this problem.

Note: You have NO permission to change directly the given HTML *(index.html file).*

Your Task

Write the missing JavaScript code to make the Central Cinema application work as expected.

Each movie has a **Name**, **Hall** and **Ticket Price**.

When you click the **[On Screen]** button and **only** if **inputs** are all **filled** and the ticket **price value** is a **number**, then a new **list item** should be added to the **Movies on Screen** section. Clear the inputs.
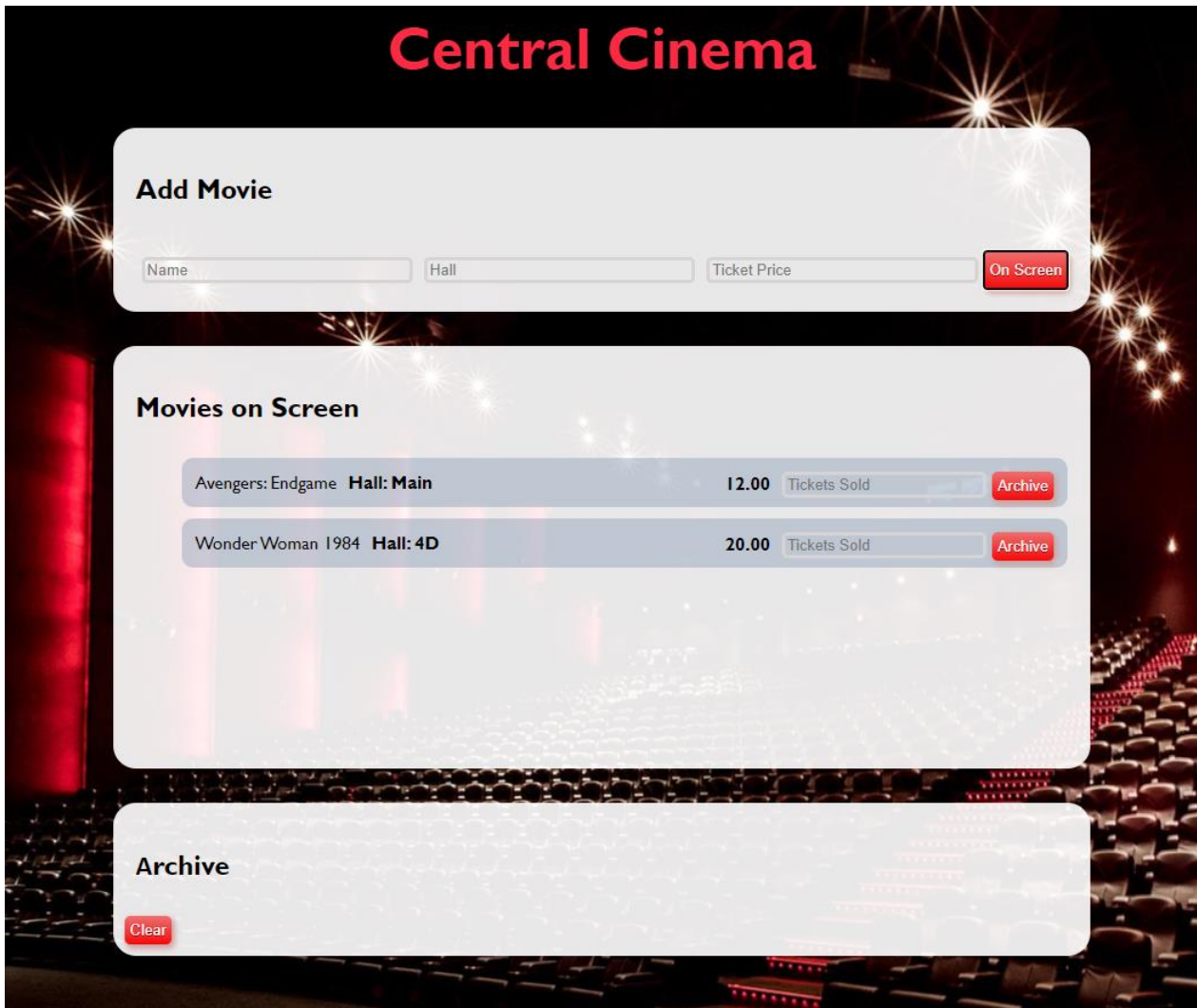
The new item should have the **following structure**:

You should create a **li** element that contains a **span** element with the name of the movie, a **strong** element with the name of the hall like: `Hall: ${hallName}`, and a **div** element. Inside the **div** element, there is a **strong**

Follow us:

element with the ticket price **(fixed to the second digit** after the decimal point), an **input** element with a **placeholder** containing: "**Tickets Sold**" and a button **[Archive].**
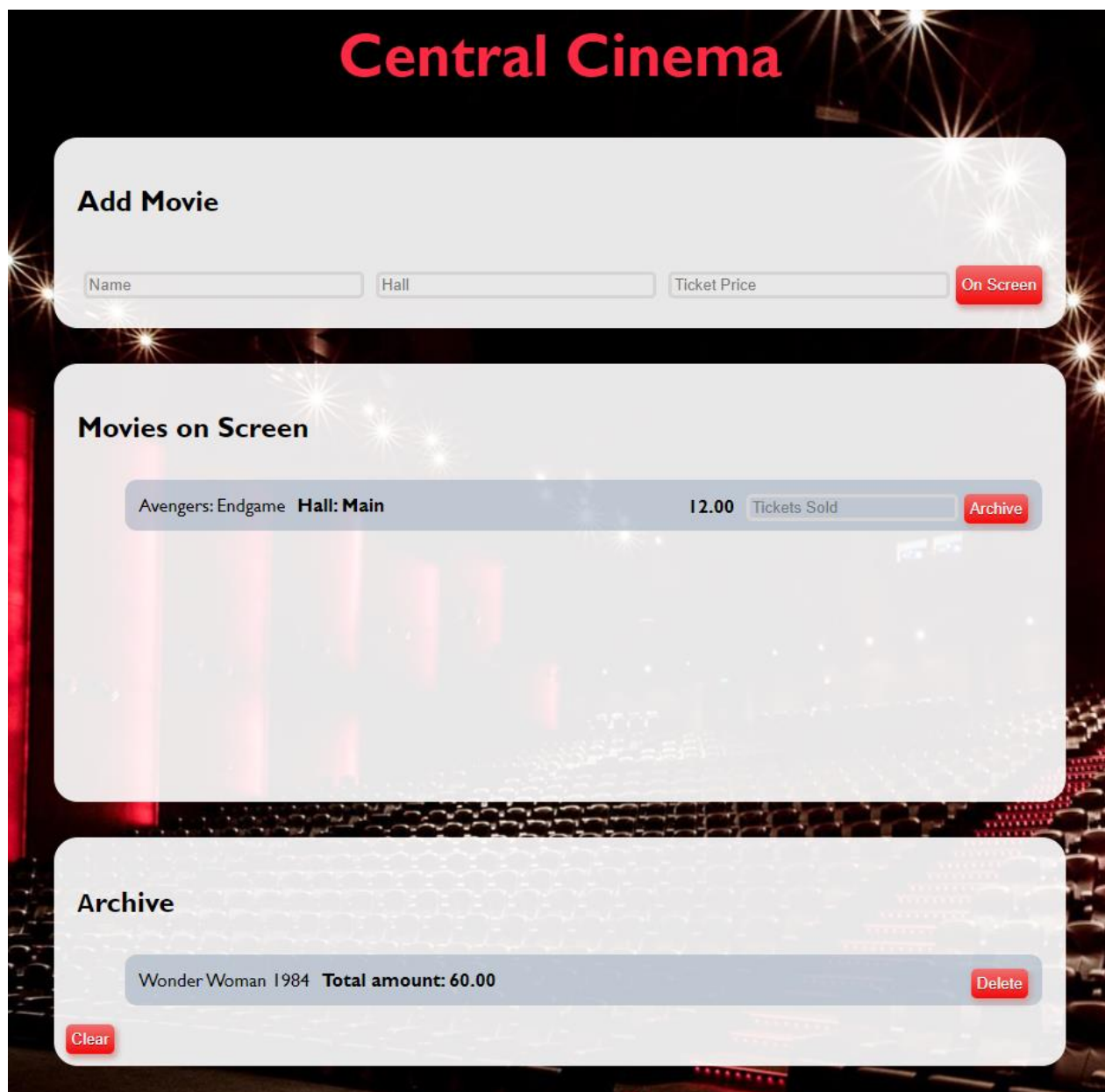
When you click the **[Archive]** button and **only** if the input for tickets count is **filled with a number** like:

You should **add** that item to the **Archive** section like a list item in the **ul,** calculating the total profit of the movie like this:

Use the following format:

Follow us:

```
▼<section id="archive">
    <h2>Archive</h2>
    ▼<ul>
        ▼<li> == $0
            <span>Wonder Woman 1984</span>
            <strong>Total amount: 60.00</strong>
            <button>Delete</button>
        </li>
    </ul>
    <button>Clear</button>
</section>
```
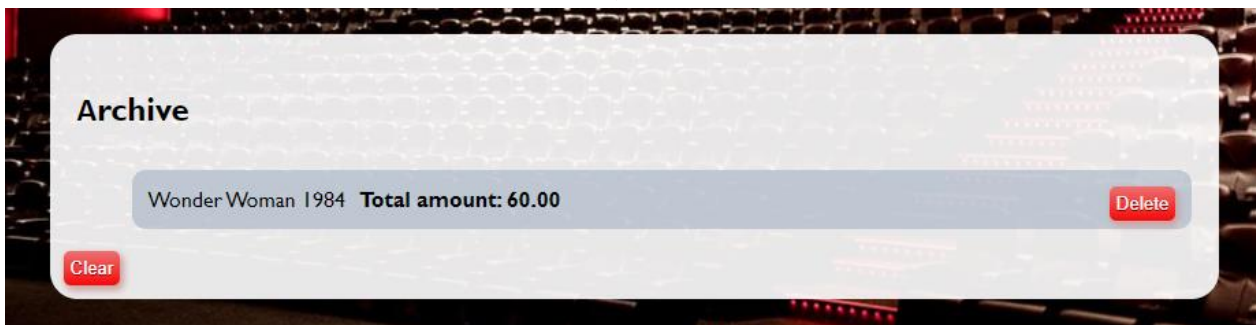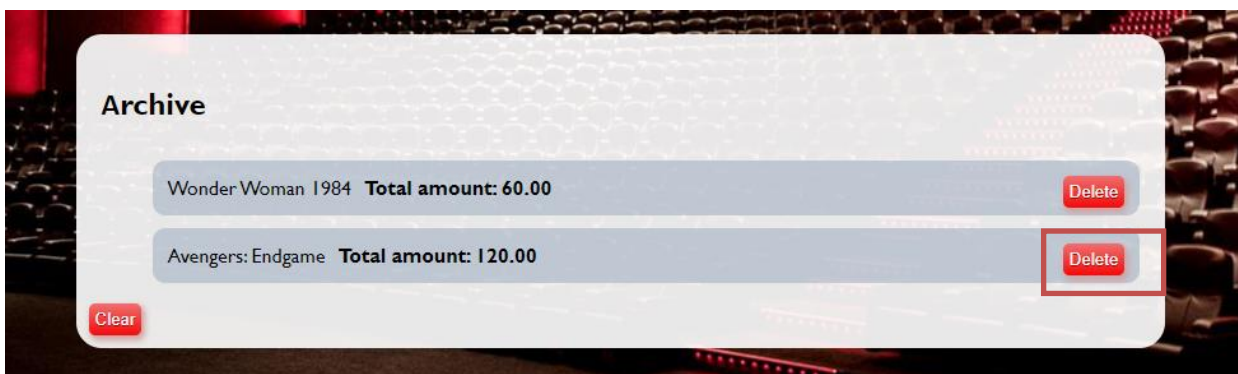
Here we have a **list item** containing **span** element with the name of the movie, **strong** element with total profit like: `` `Total amount: ${total price}` ``, **fixed to the second digit** after the decimal point. Add a delete button **[Delete]**.

When you click the **[Delete]** button, you should **delete the current list item.**

Finally, when we click the **[Clear]** button **delete** all **li** elements from the **archive** section. No matter how many archived movies we have the archive section leaves like this: