# JS Advanced: Exam Preparation

# Problem 1. Furniture Store (DOM Manipulation)

**Link in Judge:** <https://judge.softuni.org/Contests/Practice/Index/3089#0>

**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()**)
* **replaceWith()** in Judge
* **replaceAll()** in Judge
* **closest()** in Judge

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

**Use the provided skeleton to solve this problem.**

**Note**: You **can't** and you have no permission to **change** directly the given html code (index.html file).

**Write the missing JavaScript code** to make the **Furniture** **Store** work as expected:



**Your Task**

* A**ll fields (model, year, description, and price)** are **filled with the correct input**
  + **Model and description** are **non**-**empty** **strings**
  + **Year** and **Price** need to be **positive** **numbers**
  + **All fields must be filled**

1. **Getting the furniture information**



* When you click the “Add” button, the information from the input fields must be added to the table and then clear input fields.
* The table contains **Model, Price of furniture** and **Actions** - **[More information], [Buy it]**. **Price** must be **rounded** to **second** digit after decimal point.



**Each furniture** must be appended to **"furniture-list"** and look like the picture below: 

Each furniture has a main information line **(Model, Price)** and an additional information line. The additional information line stores **the description and year** of manufacture of the furniture **(hidden until the "More info" button is pressed**)**.**

When the **"More Info"** button is clicked, change button text from **"More Info"** to **"Less Info"** and style display of **"class = hide"** from **"none "** to **"contents".** The second **<td>** must have attribute **colspan** with value **3.** When click **"Less Info"** button is clicked, change button text from **"Less Info"** to **"More Info"** and style from **"contents "** to **"none".**





When the **"Buy it"** button is clicked, should have the following **functionality:**

* **The current furniture** must be **removed from the row** in the table
* You need to **change** the **total** profit in **the store.** Take the element with class **"total-price"** **and increase the current total price** **with** the **price of the furniture**.



**Problem 2. Restaurant**

**Link in Judge:** <https://judge.softuni.org/Contests/Practice/Index/3008#1>



Write a **class** **Restaurant** which has the following **functionality**:

**Constructor**

Should have 4 properties:

* **budgetMoney - number**
* **menu - object**
* **stockProducts - object**
* **history - array**

**At initialization** of the **Restaurant class**, the **constructor** accepts only the **budget!** The rest of the properties must be **empty**!

**Methods**

**loadProducts()**

Accept 1 argument **products** (**array from strings**).

* **Every element** into this array is information about product in format:

**"{productName} {productQuantity} {productTotalPrice}"**

* They are separated by a **single** **space**

**Example**: ["**Banana** **10** **5**", **"Strawberries** **50** **30", "Honey 5 50"…**]

This method **appends** **products** into our products in stock (**stockProducts**) under the following circumstances:

* **If the budget allows us to buy the current product ( {productTotalPrice} <= budget )** , we add it to **stockProducts** keeping **the name** and **quantity** of **the meal** and we **deduct** **the price of the product** from **our budget.** If the current product already exists into **stockProducts** just add the new quantity to the old one
* And finally, **whether or not** we have **added** a product to stock or **not**, we **record** our **action** in **history**:
* If we **were able to add** the current product:

"***Successfully loaded {productQuantity} {productName}***"

* If we **not**:

"***There was not enough money to load {productQuantity} {productName}***"

This method must **return all actions joined by a new line!**

**addToMenu()**

* Accept 3 arguments **meal** (string)**, needed products** (array from strings) and **price** (number).
* Every element into **needed products** is in format: **"{productName} {productQuantity}".** They are separated by a **single space**!
* If the meal is not in our menu, **appends a meal** into object **menu**. **Must have properties products and price!**
* Check how many meals have in menu and **returns one of** the following messages:
  + One meal:

***"Great idea! Now with the {meal} we have 1 meal in the menu, other ideas?"***

* Zero, Two or more meal:

***"Great idea! Now with the {meal} we have {the number of all meals in the menu} meals in the menu, other ideas?"***

* Otherwise, if we already have this meal return the **message**:

***"The {meal} is already in the our menu, try something different.*"**

**showTheMenu()**

* This method just **return** **all meals** from our **menu** **separated by a new line** in format:

{meal} - $ {meal price}

{meal} - $ {meal price}

{meal} - $ {meal price}

…

* If our menu **is empty**, just return the **message**:

"***Our menu is not ready yet, please come later...***"

**makeTheOrder()**

Accept 1 argument **meal** (string).

* This method **searches** **the menu** for a **certain meal**.
* If **we do not have** the **given meal**, **return** the following **message**:

"***There is not {meal} yet in our menu, do you want to order something else?***"

* **Otherwise,** if we **have** **this meal** in **the menu**, we need to check if we have the **needed products** to make it! If we **do not have** **all needed products** for this meal, **return** the following **message**:

**"*For the time being, we cannot complete your order ({meal}), we are very sorry...*"**

* If we **have** **this meal in the menu** and also, we **have** **all needed products** to make it, **return** the following message:

"***Your order ({meal}) will be completed in the next 30 minutes and will cost you {the current price of the meal}."***

* You also **need to reduce quantity of all used products** from those in stock and **add the price** of the meal to the **total budget**.

**Examples**

|  |
| --- |
| **Input 1** |
| **let *kitchen*** = **new** Restaurant(1000); ***console***.log(***kitchen***.loadProducts([**'Banana 10 5'**, **'Banana 20 10'**, **'Strawberries 50 30'**, **'Yogurt 10 10'**, **'Yogurt 500 1500'**, **'Honey 5 50'**])); |

|  |
| --- |
| **Output 1** |
| Successfully loaded 10 Banana Successfully loaded 20 Banana Successfully loaded 50 Strawberries Successfully loaded 10 Yogurt There was not enough money to load 500 Yogurt Successfully loaded 5 Honey |

|  |
| --- |
| **Input 2** |
| **let *kitchen*** = **new** Restaurant(1000);  ***console***.log(***kitchen***.addToMenu(**'frozenYogurt'**, [**'Yogurt 1'**, **'Honey 1'**, **'Banana 1'**, **'Strawberries 10'**], 9.99)); ***console***.log(***kitchen***.addToMenu(**'Pizza'**, [**'Flour 0.5'**, **'Oil 0.2'**, **'Yeast 0.5'**, **'Salt 0.1'**, **'Sugar 0.1'**, **'Tomato sauce 0.5'**, **'Pepperoni 1'**, **'Cheese 1.5'**], 15.55)); |

|  |
| --- |
| **Output 2** |
| Great idea! Now with the frozenYogurt we have 1 meal in the menu, other ideas? Great idea! Now with the Pizza we have 2 meals in the menu, other ideas? |

|  |
| --- |
| **Input 3** |
| **let *kitchen*** = **new** Restaurant(1000);  ***console***.log(***kitchen***.showTheMenu()); |

|  |
| --- |
| **Output 3** |
| frozenYogurt - $ 9.99 Pizza - $ 15.55 |

|  |
| --- |
| **Input 4** |
| **let *kitchen*** = **new** Restaurant(1000);  ***kitchen***.loadProducts([**'Yogurt 30 3'**, **'Honey 50 4'**, **'Strawberries 20 10'**, **'Banana 5 1'**]);  ***kitchen***.addToMenu(**'frozenYogurt'**, [**'Yogurt 1'**, **'Honey 1'**, **'Banana 1'**, **'Strawberries 10'**], 9.99);  ***console***.log(***kitchen***.makeTheOrder(**'frozenYogurt'**)); |

|  |
| --- |
| **Output 4** |
| Your order (frozenYogurt) will be completed in the next 30 minutes and will cost you 9.99. |

**Problem 3. Cinema (Unit Testing)**

**Link in Judge:** <https://judge.softuni.org/Contests/Practice/Index/3089#2>

**Your Task**

Using **Mocha** and **Chai** write **JS Unit Tests** to test a variable named **cinema**, which represents an object. You may use the following code as a template:

|  |
| --- |
| describe(**"*Tests* …"**, **function**() {  describe(**"*TODO* …"**, **function**() {  ***it***(**"*TODO …*"**, **function**() {  *//* ***TODO:*** …  });  });  *//* ***TODO:*** …  }); |

The object that should have the following functionality:

* **showMovies(movieArr)**-A function that accepts an array:
* The array includes the available movies in the cinema ([‘King Kong’, ‘The Tomorrow War’, ‘Joker’,etc.])
* If the length of the input array is zero, the function returns the string: "**There are currently no movies to show.**"
* Otherwise, the function returns: an array of available movies, separated by a comma and space
* There is no need for validation for the input, you will always be given an array
* **ticketPrice(projectionType)**- A function that accept string:
  + The function checks whether the submitted projectionType is present in the schedule with the types of projections
  + If present in the schedule, return the price according to the type
* Otherwise, the function throws an error in the following format "**Invalid projection type.**"
* There is no need for validation for the input
* **swapSeatsInHall(firstPlace, secondPlace)**- A function that accepts two numbers
* The function swaps the seat number in the cinema hall, when two numbers are submitted.
* The exchange is not successful and the function returns "**Unsuccessful** **change of seats in the hall.**" :
  + If one of the two numbers do not exist
  + The numbers are not integers
  + If one of the numbers is greater than the capacity of the hall – **20**
  + Seats are less or equal of **0**
* Otherwise return: "**Successful change of seats in the hall.**"
* There is a need for validation for the input

**JS Code**

To ease you in the process, you are provided with an implementation which meets all of the specification requirements for the **cinema** object:

|  |
| --- |
| cinema.js |
| const cinema = {  showMovies: function(movieArr) {  if (movieArr.length == 0) {  return 'There are currently no movies to show.';          } else{           let result = movieArr.join(', ');              return result;          }      },    ticketPrice: function(projectionType) {    const schedule = {              "Premiere": 12.00,              "Normal": 7.50,              "Discount": 5.50          }          if (schedule.hasOwnProperty(projectionType)) {              let price = schedule[projectionType];              return price;          } else {              throw new Error('Invalid projection type.')          }      },    swapSeatsInHall: function(firstPlace, secondPlace) {          if (!Number.isInteger(firstPlace) || firstPlace <= 0 || firstPlace > 20 ||  !Number.isInteger(secondPlace) || secondPlace <= 0 || secondPlace > 20 || firstPlace === secondPlace) {              return "Unsuccessful change of seats in the hall.";          } else {              return "Successful change of seats in the hall.";          }      }  }; |

**Submission**

Submit your tests inside a **cinema()** statement, as shown above.

*GOOD LUCK… ☺*