# More Exercises: Objects & Classes

Problems for exercises and homework for the ["JavaScript Advanced" course @ SoftUni](https://softuni.bg/courses/js-advanced). Submit your solutions in the SoftUni judge system at <https://judge.softuni.bg/Contests/2434/More-Exercise-Objects-Classes>.

## Rectangle

Write a **class** for a rectangle object. It needs to have a **width** (Number), **height** (Number) and **color** (String) properties, which are set from the constructor and a calcArea() method, that calculates and **returns** the rectangle’s area.

### Input

The constructor function will receive valid parameters.

### Output

The calcArea() method should **return** a number.

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

### Examples

|  |  |
| --- | --- |
| Sample Input | Output |
| let rect = new Rectangle(4, 5, 'red');  console.log(rect.width);  console.log(rect.height);  console.log(rect.color);  console.log(rect.calcArea()); | 4  5  Red  20 |

## Score to HTML

You are given a JSON string representing an array of objects, parse the JSON and create a table using the supplied objects. The table should have 2 columns **"name"** and **"score"**, each object in the array will also have these keys.

Any text elements **(&,<,>,",')** must also be **escaped** in order to ensure no dangerous code can be passed.

### Input

The **input** comes as array with a single string argument (the array of objects as a JSON).

### Output

The **output** should be printed on the console - a table with 2 columns - **"name"** and **"score"**, containing the values from the objects as rows.

|  |  |
| --- | --- |
| **Input** | **Output** |
| ['[{"name":"Pesho","score":479},  {"name":"Gosho","score":205}]'] | <table>  <tr><th>name</th><th>score</th></tr>  <tr><td>Pesho</td><td>479</td></tr>  <tr><td>Gosho</td><td>205</td></tr>  </table> |
| ['[{"name":"Pesho & Kiro",  "score":479  },  {"name":"Gosho, Maria & Viki",  "score":205  }]'] | <table>  <tr><th>name</th><th>score</th></tr>  <tr><td>Pesho &amp; Kiro</td><td>479</td></tr>  <tr><td>Gosho, Maria &amp; Viki</td><td>205</td></tr>  </table> |

## Count Words in a Text

You are tasked to count the number of words in a text using an object as an associative array, any combination of letters, digits and \_ (underscore) should be counted as a word. The words should be stored in the object as properties - the **key** being the **word** and the **value** being the **amount of times the word is contained** **in the text**.

### Input

The **input** comes as an array of strings containing one entry - the text whose words should be counted. The text may consist of more than one sentence.

### Output

The **output** should be printed on the console - the JSON representation of the object containing the words.

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| ['Far too slow, you're far too slow.'] | {"Far":1,  "too":2,  "slow":2,  "you":1,  "re":1,  "far":1} |
| ['JS devs use Node.js for server-side JS.-- JS for devs'] | {"JS":3,  "devs":2,  "use":1,  "Node":1,  "js":1,  "for":2,  "server":1,  "side":1} |

## City Markets

You have been tasked to follow the sales of products in the different towns. For every town you need to keep track of all the products sold, and for every product, the amount of total income.

The **town** and **product** are both **strings**. The **amount of sales** and **price for one unit** will be **numbers**. Store all towns, for every town, store its products, and for every product, its amount of **total income**. The total income is calculated with the following formula - **amount of sales \* price for one unit**. If you receive as input a town you already have, you should just **add** the **new product** to it.

### Input

The **input** comes as array of strings. Each element will represent data about a product and its sales. The format of input is:

{town} -> {product} -> {amountOfSales} : {priceForOneUnit}

### Output

As **output** you must print every town, its products and their total income in the following format:

“Town – {townName}

$$${product1Name} : {productTotalIncome}

$$${product2Name} : {productTotalIncome}

...”

The **order of output** for each of those entries is - by **order of entrance**.

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| ['Sofia -> Laptops HP -> 200 : 2000',  'Sofia -> Raspberry -> 200000 : 1500',  'Sofia -> Audi Q7 -> 200 : 100000',  'Montana -> Portokals -> 200000 : 1',  'Montana -> Qgodas -> 20000 : 0.2',  'Montana -> Chereshas -> 1000 : 0.3'] | Town - Sofia  $$$Laptops HP : 400000  $$$Raspberry : 300000000  $$$Audi Q7 : 20000000  Town - Montana  $$$Portokals : 200000  $$$Qgodas : 4000  $$$Chereshas : 300 |

## Unity

Rats are uniting.

Create a class Rat, which holds the functionality to unite with other objects of the same type. Make it so that the object holds all the other objects it has connected to.

The class should have a **name**, which is a **string**, and it should be **initialized with it**.

The class should also hold a function unite(otherRat), which unites the **first object** with the **given one**. An object should store all of the objects it has united to. The function should only add the object if it is an object of the class Rat. In any other case it should **do nothing**.

The class should also hold a function getRats() which returns all the rats it has united to, in a list.

Implement functionality for toString() function… which returns a string representation of the object and all of the objects its united with, each on a new line. On the first line put the object’s name and on the next several lines put the united objects’ names, each with a padding of “**##**”.

### Example

|  |
| --- |
| unity.js |
| **let *firstRat*** = **new** Rat(**"Peter"**); ***console***.log(***firstRat***.toString()); *// Peter* ***console***.log(***firstRat***.getRats()); *// []* ***firstRat***.unite(**new** Rat(**"George"**)); ***firstRat***.unite(**new** Rat(**"Alex"**)); ***console***.log(***firstRat***.getRats()); *// [ Rat { name: 'George', unitedRats: [] }, // Rat { name: 'Alex', unitedRats: [] } ]* ***console***.log(***firstRat***.toString()); *// Peter // ##George // ##Alex* |

### Hints

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

## Usernames

You are tasked to create a catalogue of usernames. The usernames will be strings that **may contain any ASCII** character. You **need to order** them **by their length**, in **ascending order**, as **first criteria**, and by **alphabetical order** as **second criteria**.

### Input

The **input** comes as array of strings. Each element represents a **username**. Sometimes the input may contain **duplicate usernames**. Make it so that there are **NO duplicates** in the output.

### Output

The **output** is all of the usernames, **ordered** exactly as **specified above** – each printed on a new line.

### Examples

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Input** | **Output** |  | **Input** | **Output** |
| ['Ashton', 'Kutcher', 'Ariel', 'Lilly', 'Keyden', 'Aizen', 'Billy', 'Braston'] | Aizen Ariel Billy Lilly Ashton Keyden Braston Kutcher |  | ['Denise', 'Ignatius', 'Iris', 'Isacc', 'Indie', 'Dean', 'Donatello', 'Enfuego', 'Benjamin', 'Biser', 'Bounty', 'Renard', 'Rot'] | Rot Dean Iris Biser Indie Isacc Bounty Denise Renard Enfuego Benjamin Ignatius Donatello |

### Hints

* Try to find a **structure** which **does NOT allow duplicates**, it will be best for the current problem.

## Unique Sequences

You are tasked with storing sequences of numbers. You will receive an unknown amount of **arrays containing numbers** from which you must store only the **unique** arrays (duplicate arrays should be discarded). An array is considered the **same** (**NOT unique**) if it contains the **same numbers** as another array**, regardless of their order**.

After storing all arrays, your program should print them back in **ascending** order based on their **length**, if two arrays have the same length they should be printed in **order of being received from the input**. Each individual array should be printed in **descending order** in the format **"[a1, a2, a3,… an]"**. Check the examples bellow.

### Input

The **input** comes as an array of strings where each entry is a JSON representing an array of numbers.

### Output

The **output** should be printed on the console - each array printed on a new line in the format **"[a1, a2, a3,… an]"** , following the above mentioned ordering.

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| ["[-3, -2, -1, 0, 1, 2, 3, 4]",  "[10, 1, -17, 0, 2, 13]",  "[4, -3, 3, -2, 2, -1, 1, 0]"] | [13, 10, 2, 1, 0, -17]  [4, 3, 2, 1, 0, -1, -2, -3] |

|  |  |
| --- | --- |
| **Input** | **Output** |
| ["[7.14, 7.180, 7.339, 80.099]",  "[7.339, 80.0990, 7.140000, 7.18]",  "[7.339, 7.180, 7.14, 80.099]"] | [80.099, 7.339, 7.18, 7.14] |

### Hints

* Think of an easy way to compare arrays.
* Sometimes the most obvious collection choice is not the best one.

## Cards

You need to write an **IIFE** that results in an object containing two properties Card which is a class and Suits which is an object that will hold the possible suits for the cards.

The Suits object should have exactly these 4 properties:

* **SPADES**: ♠
* **HEARTS**: ♥
* **DIAMONDS**: ♦
* **CLUBS**: ♣

Where the key is **SPADES**, **HEARTS** e.t.c. and the value is the actual symbol ♠, ♥ and so on.

The Card class should allow for creating cards, each card has 2 properties **face** and **suit**. The **valid** faces are the following ["2", "3", "4", "5", "6", "7", "8", "9", "10", "J", "Q", "K", "A"] any other are considered invalid.

The Card class should have setters and getters for the **face** and **suit** properties, when creating a card or setting a property validations should be performed, if an invalid face or a suit not in the Suits object is passed an Error should be **thrown**.

### Code Template

You are required to write and submit an **IIFE** which results in an object containing the above-mentioned Card and Suits as properties. Here is an example template you can use:

|  |
| --- |
| cards.js |
| (**function**(){  *//* ***TODO:***  **return** {  **Suits**:***Suits***,  **Card**:***Card*** } }()) |

### Screenshot

An example usage should look like this:



## Instance Validation

Write a class for a checking account that validates it’s created with valid parameters. A CheckingAccount has a clientId, email, firstName, lastName. Each parameter must meet specific requirements:

* clientId - Must be a string representing a **6-digit number**; if invalid, throw a TypeError with the message "Client ID must be a 6-digit number"
* email - Must contain at least one **alphanumeric** **character**, followed by the **@** symbol, followed by **one** or **more** letters or periods; all letters must be **Latin**; if invalid, throw a TypeError with message "Invalid e-mail"
* firstName, lastName - Must be at least **3** and at most **20** characters long, containing **only** Latin letters;
  + If the **length** is invalid, throw a TypeError with message:

"{****First****/****Last****} name must be between 3 and 20 characters long"

* + If invalid **characters** are used, throw a TypeError with message:

"{First/Last} name must contain only Latin characters" (replace First/Last with the relevant word)

All checks must happen in the **order** in which **they are listed** - if more than one parameter is **invalid**, throw an error for the first encountered. Note that **error messages** must be **exact**.

Submit your solution containing a single class definition.

### Examples

|  |
| --- |
| Sample Input |
| let acc = new CheckingAccount('1314', 'ivan@some.com', 'Ivan', 'Petrov') |
| Output |
| TypeError: Client ID must be a 6-digit number |

|  |
| --- |
| Sample Input |
| let acc = new CheckingAccount('131455', 'ivan@', 'Ivan', 'Petrov') |
| Output |
| TypeError: Invalid e-mail |

|  |
| --- |
| Sample Input |
| let acc = new CheckingAccount('131455', 'ivan@some.com', 'I', 'Petrov') |
| Output |
| TypeError: First name must be between 3 and 20 characters long |
| Sample Input |
| let acc = new CheckingAccount('131455', 'ivan@some.com', 'Ivan', 'P3trov') |
| Output |
| TypeError: "First name must contain only Latin characters |

## \*Extensible Class

Your task here is to deliver a **class** that can be **extended**. Implement an extend(template) method that would copy **all** of the properties of **template** to the **instance** (not to all instances, just the one from which the method was called) and if the property is a function, add it to the object’s **prototype** instead.

In addition, the base class needs to have an **ID property** that is **unique** and **autoincremented** sequentally for every new instance.

### Input / Output

The extend() function of your **class** will receive a valid object as **input parameter**, and has **no** output.

Structure your code as an **IIFE** that **returns** the class.

### Examples

|  |  |
| --- | --- |
| Sample Input | Output |
| let obj1 = new Extensible();  let obj2 = new Extensible();  let obj3 = new Extensible();  console.log(obj1.id);  console.log(obj2.id);  console.log(obj3.id); | 0  1  2 |

|  |  |
| --- | --- |
| Extensible object | Resulting object |
| obj1: {  \_\_proto\_\_: {  extend: function () {…}  },  id: 0  } | myObj: {  \_\_proto\_\_: {  extend: function () {…},  extensionMethod: function () {…}  },  id: 0,  extensionProperty: 'someString'  } |
| Template object |
| template: {  extensionMethod: function () {…},  extensionProperty: 'someString'  } |

### Hints

You may have to keep track of the last assigned ID in a **closure** that is accessible by the constructor. Constructor functions offer direct access to their prototypes - you can view and modify them with className.prototype.

## \*Arena Tier

Pesho is a pro gladiator, he is struggling to become master of the Arena.

You will receive **several input lines** in one of the following formats:

"{gladiator} -> {technique} -> {skill}"

"{gladiator} vs {gladiator}"

The gladiator and technique are strings, the given **skill** will be an integer number. You need to keep track of **every gladiator**.

When you receive a **gladiator and his technique and skill**, add him to the gladiator pool, if he isn`t present, else add his technique or update his skill, only if the current technique skill is lower than the new value.

If you receive **"{gladiator} vs {gladiator}"** and both gladiators exist in the tier, they duel with the following rules:

Compare their techniques, if they got at least one in common, the gladiator with better total skill points wins and the other is demoted from the tier -> remove him.

If they don't have techniques in common, the duel isn`t happening and both continue in the Season.

You should end your program when you receive the command "Ave Cesar". At that point you should print the gladiators, **ordered by total skill in desecending order, then ordered by name in ascending order**. Foreach gladiator print their technique and skill, **ordered desecending, then ordered by technique name in ascending order**

### Input / Constraints

You will receive an **array of strings** as a parameter to your solution.

* The input comes in the form of commands in one of the formats specified above.
* Gladiator and technique **will always be one word string, containing no whitespaces**.
* Skill will be an **integer** in the **range [0, 1000]**.
* There will be **no invalid** input lines.
* The programm ends when you receive the command "Ave Cesar".

### Output

* The output format for each gladiator is:

"{gladiator}: {totalSkill} skill"

"- {technique} <!> {skill}"

***Scroll down to see examples.***

### Examples

|  |  |  |
| --- | --- | --- |
| **Input** | **Output** | **Comments** |
| Pesho -> BattleCry -> 400  Gosho -> PowerPunch -> 300  Stamat -> Duck -> 200  Stamat -> Tiger -> 250  Ave Cesar | Stamat: 450 skill  - Tiger <!> 250  - Duck <!> 200  Pesho: 400 skill  - BattleCry <!> 400  Gosho: 300 skill  - PowerPunch <!> 300 | We order the gladiators by total skill points descending, then by name. We print every technique along its skill ordered descending by skill, then by technique name. |
| **Input** | **Output** |  |
| Pesho -> Duck -> 400  Julius -> Shield -> 150  Gladius -> Heal -> 200  Gladius -> Support -> 250  Gladius -> Shield -> 250  Pesho vs Gladius  Gladius vs Julius  Gladius vs Gosho  Ave Cesar | Gladius: 700 skill  - Support <!> 250  - Shield <!> 250  - Heal <!> 200  Pesho: 400 skill  - Duck <!> 400 | Gladius and Pesho don`t have common technique, so the duel isn`t valid.  Gladius wins vs Julius /common technique: "Shield". Julius is demoted.  Gosho doesn`t exist so the duel isn`t valid.  We print every gladiator left in the tier. |

## \*Game of Epicness

Write a JavaScript program that **determines** the **winner** from **all battles**. You will receive **two** arguments:

The **first** argument is an **array of kingdoms with generals and their army** in the form of an **object** with format:

{ kingdom: String, general: String, army: Number }

Every **general** has his own **army** that fights for a certain **kingdom**. Note that, every **kingdom’s name** is **unique,** and every general’s **name** is **unique** in **this kingdom**. If the **general** already **exists** **in** this **kingdom** **add** the **army** to his current one. After you go through all the kingdoms with their generals with armies and store the information about them, it’s time to start the battles.

The **second** argument is **matrix of strings** showing which **kingdom’s generals** are **fighting** in this format:

**[**

**[ "{AttackingKingdom} ", "{AttackingGeneral}", "{DefendingKingdom} ", "{DefendingGeneral}" ],**

**…**

**]**

The **first two elements** are the **names** of the **attacking general from** certain **kingdom** and the **second two** are the **names** of the **defending general from** certain **kingdom**. **Compare** the two general’s **armies to determine** who **wins** and who **losses** based on who have the **larger army wins.** The **winner’s army increases** with **10%** and the **loser’s army decreases** with **10%.** Keep in mind to **round** them **down** if there is any excess **army** **after the battle.** If there is a **draw**, **do not do anything**. **Keep track** of the **wins** and **losses** for every general’s battle.

Note that, **generals** from the **same kingdom** **cannot** **attack** **each other**.

After you finish with all battles you need to **find** which **kingdom** **wins** the game. To decide that, **first** **order them** by all their **general’s wins (descending)** then by their **losses (ascending),** and finally by the **kingdom’s name** in **ascending alphabetical** order.

### Input

You will receive **two arguments –** an **array of objects** with properties and a **matrix of strings** as shown above.

### Output

Print on the **console** the winning kingdom and **sort** the generals by their **armies in** **descending** order, **formatted** as seen in the examples.

### Constraints

* The **number** of **elements** in the **first input argument** will be in range **[1..100] inclusive**
* The **number** of **elements** in the **second input argument** will be in range **[0..100] inclusive**
* General’s **army** will be always an **integer** in range **[0..1,000,000] inclusive**
* There **will** be **no invalid** **input**
* There **will** be **no matching number** of **armies** in the **output**

### Examples

|  |
| --- |
| **Input** |
| [ { kingdom: "Maiden Way", general: "Merek", army: 5000 },  { kingdom: "Stonegate", general: "Ulric", army: 4900 },  { kingdom: "Stonegate", general: "Doran", army: 70000 },  { kingdom: "YorkenShire", general: "Quinn", army: 0 },  { kingdom: "YorkenShire", general: "Quinn", army: 2000 },  { kingdom: "Maiden Way", general: "Berinon", army: 100000 } ],  [ ["YorkenShire", "Quinn", "Stonegate", "Ulric"],  ["Stonegate", "Ulric", "Stonegate", "Doran"],  ["Stonegate", "Doran", "Maiden Way", "Merek"],  ["Stonegate", "Ulric", "Maiden Way", "Merek"],  ["Maiden Way", "Berinon", "Stonegate", "Ulric"] ] |
| **Output** |
| Winner: Stonegate  /\general: Doran  ---army: 77000  ---wins: 1  ---losses: 0  /\general: Ulric  ---army: 5336  ---wins: 2  ---losses: 1 |
| **Explanation** |
| After you successfully store the kingdoms information, the first battle’s result is victory for the defender Ulric and a loss for the attacker Quinn. Second battle is ignored because the generals are from the same kingdom. Third battle is a victory for Doran and a loss for Merek. Fourth battle is a win for Ulric and a loss for Merek. Fifth battle is a victory for Berinon and a defeat for Ulric. All winners increase their armies with 10% for each win and all losers decrease their armies with 10% for each loss.  The result from the battles are – Stonegate: 3 wins and 1 loss; Maiden Way: 1 win and 2 losses; YorkenShire: 0 wins and 1 loss. Making Stonegate the winner of the games because they have the most wins from kingdoms. |

|  |
| --- |
| **Input** |
| [ { kingdom: "Stonegate", general: "Ulric", army: 5000 },  { kingdom: "YorkenShire", general: "Quinn", army: 5000 },  { kingdom: "Maiden Way", general: "Berinon", army: 1000 } ],  [ ["YorkenShire", "Quinn", "Stonegate", "Ulric"],  ["Maiden Way", "Berinon", "YorkenShire", "Quinn"] ] |
| **Output** |
| Winner: YorkenShire  /\general: Quinn  ---army: 5500  ---wins: 1  ---losses: 0 |
| **Explanation** |
| The first battle between Quinn and Ulric is a draw because they have even armies because of that it is not recorded and their armies size does not change. The second battle is a win for Quinn and a loss for Berinon making YorkenShire the winner of the game with 1 win and 0 losses. |

|  |
| --- |
| **Input** |
| [ { kingdom: "Maiden Way", general: "Merek", army: 5000 },  { kingdom: "Stonegate", general: "Ulric", army: 4900 },  { kingdom: "Stonegate", general: "Doran", army: 70000 },  { kingdom: "YorkenShire", general: "Quinn", army: 0 },  { kingdom: "YorkenShire", general: "Quinn", army: 2000 } ],  [ ["YorkenShire", "Quinn", "Stonegate", "Doran"],  ["Stonegate", "Ulric", "Maiden Way", "Merek"] ] |
| **Output** |
| Winner: Maiden Way  /\general: Merek  ---army: 5500  ---wins: 1  ---losses: 0 |

## JS Advanced – Retake Exam: 18.11.2018

## 13. \*\*Kitchen



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

### Constructor

Should have 4 properties:

* budget
* menu
* productsInStock
* actionsHistory

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

### Methods:

* **LoadProducts()**
* Accept 1 property **products** (**array from strings**).
  + **Every element** into this array is information about product **in format**:  
    **"{productName} {productQuantity} {productPrice}"**
  + 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 (**productsInStock**) under the following circumstances:
* **If the budget allows us to buy the current product**, we add it to **productsInStock** 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 **productsInStock** just add the new quantity
* And finally, **whether or not** we have **added** a product to stock or **not**, we **record** our **action** in the **actionsHistory**:
* 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 properties **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**!
* This method **appends a new meal** into our **menu and returns** the following message:

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

* **If** we **do not have** the **given meal** into our **menu**, we added it **keeping** **all** that we are given as information. Otherwise if we already have this meal print the **message**:

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

* **ShowTheMenu()**
* This method just **prints** **all meals** from our **menu** **separated by a new line** in format:

**{meal} - $ {meal price}**

**{meal} - $ {meal price}**

**{meal} - $ {meal price}**

**…**

At the end **trim the result!**

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

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

* **MakeTheOrder()**
* Accept 1 property **meal** (string).
* This method **searches** **the menu** for a **certain meal**.
* If **we do not have** the **given meal**, print 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, print 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, print 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 remove all used products** from those in stock and **add the price** of the meal to the **total budget**.

### Submission

Submit only the **Kitchen** **class** as **JavaScript code**.

### Examples

|  |
| --- |
| Sample Input |
| let *kitchen* = new Kitchen (1000); *console*.log(*kitchen*.loadProducts(['Banana 10 5', 'Banana 20 10', 'Strawberries 50 30', 'Yogurt 10 10', 'Yogurt 500 1500', 'Honey 5 50'])); |
| Output |
| 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 |
| Sample Input |
| *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 |
| Great idea! Now with the frozenYogurt we have 1 meals on the menu, other ideas? Great idea! Now with the Pizza we have 2 meals on the menu, other ideas? |
| Sample Input |
| *console*.log(*kitchen*.showTheMenu()); |
| Output |
| frozenYogurt - $ 9.99 Pizza - $ 15.55 |