# Exercise: Classes & Unit Testing on Classes

Submit your solutions in the SoftUni judge system at:

<https://judge.softuni.org/Contests/2769/Classes-and-Attributes-Exercise>

# Classes

## Rectangle

Write a **class** **Rectangle** 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 |

## Data Class

Write a **class** **Request** 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 through 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:



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 this identifier:



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:



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 this identifier tells the interpreter to look for a variable in a different context, so this.method is not the same as the method.

Our class is complete and can be submitted to [Judge](https://judge.softuni.bg/Contests/Practice/Index/1534#0).

## 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 (the 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 were 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' } ] |

## Sorted List

Implement a **class List**, which **keeps** a list of numbers, sorted in **ascending order**. It must support the following functionality:

* add(element) - 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 functions 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 its result.  
Your **add** and **remove** **functions** should **return** a **class** **instance** with the required functionality as its 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 |

## Length Limit

Create a class Stringer, which holds the **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 functional 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 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.

## Company

**class** Company {  
 *//* ***TODO: implement this class...***  
}

Write a class **Company**, which following these requirements:

The **constructor** takes no parameters:

You could initialize an object:

* **departments** - empty object

addEmployee({name}, {salary}, {position}, {department})

This function should add a new employee to the **department with the given name**.

* If one of the passed parameters is an empty string (""), undefined or null, this function should **throw** an **error** with the following message: **"Invalid input!"**
* If salary is less than 0, this function should **throw** an **error** with the following message: **"Invalid input!"**
* If the new employee is hired successfully, you should add him into the **departments** object with the current name of the department and return the following message: **`New employee is hired. Name: {name}. Position: {position}`**

bestDepartment()

This functionshould return the **department** with the **highest average salary rounded** to the second digit after the decimal point and its **employees sorted** by their **salary** by **descending** order and by their **name** in **ascending** order as a second criterion:

**`Best Department is: {best department's name}**

**Average salary: {best department's average salary}**

**{employee1} {salary} {position}**

**{employee2} {salary} {position}**

**{employee3} {salary} {position}**

**…`**

### Submission

Submit only the **Company** class definition.

### Examples

This is an example of how the code is **intended to be used**:

|  |
| --- |
| Sample code usage |
| let c = new Company();  c.addEmployee("Stanimir", 2000, "engineer", "Construction");  c.addEmployee("Pesho", 1500, "electrical engineer", "Construction");  c.addEmployee("Slavi", 500, "dyer", "Construction");  c.addEmployee("Stan", 2000, "architect", "Construction");  c.addEmployee("Stanimir", 1200, "digital marketing manager", "Marketing");  c.addEmployee("Pesho", 1000, "graphical designer", "Marketing");  c.addEmployee("Gosho", 1350, "HR", "Human resources");  console.log(c.bestDepartment()); |
| Corresponding output |
| Best Department is: Construction  Average salary: 1500.00  Stan 2000 architect  Stanimir 2000 engineer  Pesho 1500 electrical engineer  Slavi 500 dyer |

## HEX

**class** Hex {  
 *//* ***TODO: implement this class...***  
}

Write a class **Hex**, having the following functionality:

* The **constructor** takes one parameter **value**, which is a number
* valueOf() This function should return the **value** property of the hex class.
* toString() This functionwill show its **hexadecimal value** starting with **"0x"**
* plus({number}) This function should add a number or Hex object and return a new Hex object.
* minus({number}) This function should subtract a number or Hex object and return a new Hex object.
* parse({string}) Create a **parse class method** that can **parse** Hexadecimal numbers and convert them to standard decimal numbers.

### Submission

Submit only your **Hex class.**

### Examples

This is an example of how the code is **intended to be used**:

|  |  |
| --- | --- |
| **Input** | **Output** |
| let FF = new Hex(255);  console.log(FF.toString());  FF.valueOf() + 1 == 256;  let a = new Hex(10);  let b = new Hex(5);  console.log(a.plus(b).toString());  console.log(a.plus(b).toString()==='0xF');  console.log(FF.parse('AAA')); | 0XFF  0XF  true  2730 |

# Built-in Collections

## Juice Flavors

You will be given different juices, as **strings**. You will also **receive quantity** as a **number**. If you receive a juice that 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 many 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 an 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 the **order of obtaining the bottles**. Check the second example below - 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 created Pear and Watermelon juice bottles, thus the Kiwi bottles appear last in the output.

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| ['Orange => 2000',  'Peach => 1432',  'Banana => 450',  'Peach => 600',  'Strawberry => 549'] | Orange => 2  Peach => 2 |
| ['Kiwi => 234',  'Pear => 2345',  'Watermelon => 3456',  'Kiwi => 4567',  'Pear => 5678',  'Watermelon => 6789'] | Pear => 8  Watermelon => 10  Kiwi => 4 |

# Classes Interacting with DOM

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.

## Contacts Builder

Write a JS **class** that generates a **contact** info box. You will receive a person's first name, last name, phone, and email. Compose the markup for the contact box, attach all the needed events, and when the **render** function is called, **append** the newly created element to the document.

A contact infobox is **composed** of first name, last name, phone, email (all strings), and property that indicates if the contact is online or not. Clicking a button on the box **toggles** the visibility of the person's contact information (phone and email). *See the examples for more details.*

The **constructor** of your class needs to take **four**-string arguments - first name, last name, phone, email. Additionally, the class should also contain the following functionality:

* Property online – Boolean value, initially set to **false**
* Function render(id)– **appends** the Contact's **HTML element representation** to the **DOM element** with **Id** equal to the argument

When the value of the online property is changed, the corresponding HTML should be updated – if it’s set to true, add the class "**online**" to the div with class "**title**" (containing the name). If it’s false, remove the class "**online**".

A contact info box should have the following HTML structure:

|  |
| --- |
| Contact |
| <**article**>  <**div class="title"**>{firstName lastName}<**button**>**&#8505;**</**button**></**div**>  <**div class="info"**>  <**span**>**&phone;** {phone}</**span**>  <**span**>**&#9993;** {email}</**span**>  </**div**>  </**article**> |

When the box is initially created, the div with class "**info**" must be **hidden**. Clicking the button **toggles its visibility**.

You can use the HTML skeleton to test your functionality.

**Hint:** Use <https://www.toptal.com/designers/htmlarrows/symbols/> to get Unicode escapes. (Example: Phone -> '\u260E').

### Examples

Your solution can be **tested** using the following code:

|  |
| --- |
| Sample JavaScript |
| let contacts = [  **new** Contact(**"Ivan"**, **"Ivanov"**, **"0888 123 456"**, **"i.ivanov@gmail.com"**),  **new** Contact(**"Maria"**, **"Petrova"**, **"0899 987 654"**, **"mar4eto@abv.bg"**),  **new** Contact(**"Jordan"**, **"Kirov"**, **"0988 456 789"**, **"jordk@gmail.com"**)  ];  contacts.forEach(c => c.render(**'main'**));  *// After 1 second, change the online status to true*  setTimeout(() => contacts[1].**online** = **true**, 2000); |







## View Model

We need to create a class Textbox that represents one or more **HTML input** elements with type="text". The constructor takes as parameters a **selector** and a **regex** for invalid symbols.

Textbox elements created from the class should have:

* property value (has getters and setters)
* property \_elements containing the set of elements matching the selector
* getter elements for the \_elements property – return as NodeList
* property \_invalidSymbols - a regex used for validating the textbox value
* method isValid() - if the \_invalidSymbols regex can be matched in any of the **\_elements values** return false, otherwise, return true.

All \_elements values and the value property should be linked. If the value of an element from \_elements change all other elements' values and the value property should instantly reflect it, likewise should happen if the value property changes.

### Constraints

* Selectors will always point to input elements with type text.

### Example

To help you test your code, you're provided with an **HTML** template.

And an example **JS** **skeleton**.

Here is an example output in the browser:



And the isValid function.



Submit only the Textbox class definition.

### Hints

* Pay attention to what event you use, different events trigger on different conditions. You want an event that is directly linked to changes in the value of an input element.
* Pay close attention to the value of this when writing event handler functions.

# Unit Testing on Classes

## Payment Package

You are given the following **JavaScript class**:

|  |
| --- |
| PaymentPackage.js |
| **class** PaymentPackage {  constructor(name, value) {  **this**.name = name;  **this**.value = value;  **this**.VAT = 20; *// Default value* **this**.active = **true**; *// Default value* }   **get** name() {  **return this**.**\_name**;  }   **set** name(newValue) {  **if** (**typeof** newValue !== **'string'**) {  **throw new** Error(**'Name must be a non-empty string'**);  }  **if** (newValue.length === 0) {  **throw new** Error(**'Name must be a non-empty string'**);  }  **this**.**\_name** = newValue;  }   **get** value() {  **return this**.**\_value**;  }   **set** value(newValue) {  **if** (**typeof** newValue !== **'number'**) {  **throw new** Error(**'Value must be a non-negative number'**);  }  **if** (newValue < 0) {  **throw new** Error(**'Value must be a non-negative number'**);  }  **this**.**\_value** = newValue;  }   **get** VAT() {  **return this**.**\_VAT**;  }   **set** VAT(newValue) {  **if** (**typeof** newValue !== **'number'**) {  **throw new** Error(**'VAT must be a non-negative number'**);  }  **if** (newValue < 0) {  **throw new** Error(**'VAT must be a non-negative number'**);  }  **this**.**\_VAT** = newValue;  }   **get** active() {  **return this**.**\_active**;  }   **set** active(newValue) {  **if** (**typeof** newValue !== **'boolean'**) {  **throw new** Error(**'Active status must be a boolean'**);  }  **this**.**\_active** = newValue;  }   toString() {  **const** output = [  **`Package:** ${**this**.name}**`** + (**this**.active === **false** ? **' (inactive)'** : **''**),  **`- Value (excl. VAT):** ${**this**.value}**`**,  **`- Value (VAT** ${**this**.VAT}**%):** ${**this**.value \* (1 + **this**.VAT / 100)}**`** ];  **return** output.join(**'\n'**);  } } |

### Functionality

The above code defines a **class** that contains information about a **payment package**. An **instance** of the class should support the following operations:

* Can be **instantiated** with two parameters - a string name and number value
* Accessor name - used to get and set the value of the name
* Accessor value - used to get and set the value of value
* Accessor VAT - used to get and set the value of VAT
* Accessor active - used to get and set the value of active
* Function toString() - return a string, containing an overview of the instance; if the package is **not active**, append the label "**(inactive)**" to the printed **name**

When creating an instance, or changing any of the property values, the parameters are validated. They must follow these rules:

* name - non-empty string
* value - non-negative number
* VAT - non-negative number
* active - Boolean

If any of the requirements aren’t met, the operation must throw an error.

***Scroll down for examples and details about submitting to Judge.***

### Example

This is an example of how this code is **intended to be used**:

|  |
| --- |
| Sample code usage |
| *// Should throw an error* **try** {  **const *hrPack*** = **new** PaymentPackage(**'HR Services'**); } **catch**(err) {  **console**.log(**'Error: '** + err.**message**); } **const *packages*** = [  **new** PaymentPackage(**'HR Services'**, 1500),  **new** PaymentPackage(**'Consultation'**, 800),  **new** PaymentPackage(**'Partnership Fee'**, 7000), ]; **console**.log(***packages***.join(**'\n'**));  **const *wrongPack*** = **new** PaymentPackage(**'Transfer Fee'**, 100); *// Should throw an error* **try** {  ***wrongPack***.active = **null**; } **catch**(err) {  **console**.log(**'Error: '** + err.**message**); } |
| Corresponding output |
| Error: Value must be a non-negative number  Package: HR Services  - Value (excl. VAT): 1500  - Value (VAT 20%): 1800  Package: Consultation  - Value (excl. VAT): 800  - Value (VAT 20%): 960  Package: Partnership Fee  - Value (excl. VAT): 7000  - Value (VAT 20%): 8400  Error: Active status must be a boolean |

### Your Task

Using **Mocha** and **Chai** write **unit tests** to test the entire functionality of the PaymentPackage class. Make sure instances of it have all the required functionality and validation. You may use the following code as a template:

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