

# Exercises: Classes

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/340/>.

## 1. Data Class

Write a JS class 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**.

### Input / Output

The constructor of your class will receive valid parameters. There is no output.

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

### Examples

| Sample Input  | Resulting object  |
|---|---|
| <pre>let myData = new Request('GET',<br/>'http://google.com', 'HTTP/1.1', '')</pre> | <pre>{ method: 'GET',<br/>  uri: 'http://google.com',<br/>  version: 'HTTP/1.1',<br/>  message: '',<br/>  response: undefined,<br/>  fulfilled: false }</pre> |

## 2. Tickets

Write a JS program that manages a database of tickets. A ticket has a **destination**, a **price** and a **status**. Your program will receive **two argument** – the first is an **array of strings** for ticket descriptions and the second is a **string**, representing a **sorting criteria**. 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 (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 where registered.

## Examples

| Sample Input  | Output Array   |
|---|--|
| <code>['Philadelphia 94.20 available',<br/>'New York City 95.99 available',<br/>'New York City 95.99 sold',<br/>'Boston 126.20 departed'],<br/>'destination'</code> | <code>[ Ticket { destination: 'Boston',<br/>price: 126.20,<br/>status: 'departed' },<br/>Ticket { destination: 'New York City',<br/>price: 95.99,<br/>status: 'available' },<br/>Ticket { destination: 'New York City',<br/>price: 95.99,<br/>status: 'sold' },<br/>Ticket { destination: 'Philadelphia',<br/>price: 94.20,<br/>status: 'available' } ]</code> |
| <code>['Philadelphia 94.20 available',<br/>'New York City 95.99 available',<br/>'New York City 95.99 sold',<br/>'Boston 126.20 departed'],<br/>'status'</code>      | <code>[ Ticket { destination: 'Philadelphia',<br/>price: 94.20,<br/>status: 'available' },<br/>Ticket { destination: 'New York City',<br/>price: 95.99,<br/>status: 'available' },<br/>Ticket { destination: 'Boston',<br/>price: 126.20,<br/>status: 'departed' },<br/>Ticket { destination: 'New York City',<br/>price: 95.99,<br/>status: 'sold' } ]</code> |

## 3. 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 of 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

```
test.js

let test = new Rat("Pesho");
console.log(test.toString()); //Pesho

console.log(test.getRats()); //[]

test.unite(new Rat("Gosho"));
test.unite(new Rat("Sasho"));
console.log(test.getRats());
//[ Rat { name: 'Gosho', unitedRats: [] },
// Rat { name: 'Sasho', unitedRats: [] } ]

console.log(test.toString());
// Pesho
// ##Gosho
// ##Sasho
```

## Hints

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

## 4. Length Limit

Create a class, **Stringer**, which holds **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 functionality 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 to 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

```
test.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.

## 5. Extensible Class

**Revisit** the problem Extensible Object from Object Composition. Refactor the code so that instead of a single instance, you deliver a **class** that can be **extended**. The class has 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** sequentially for every new instance.

## Input / Output

The **extend()** function of your object 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           |
|---|------------------|
| <pre>let obj1 = new Extensible(); let obj2 = new Extensible(); let obj3 = new Extensible(); console.log(obj1.id); console.log(obj2.id); console.log(obj3.id);</pre> | <pre>0 1 2</pre> |

| Extensible object  | Resulting object   |
|--|--|
| <pre>obj1: {   __proto__: {     extend: function () {...}   },   id: 0 }</pre> | <pre>myObj: {   __proto__: {     extend: function () {...},     extensionMethod: function () {...}   },   id: 0,   extensionProperty: 'someString'</pre> |
| 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**.

## 6. Sorted List

Revisit the problem Sorted List from Object Composition. Refactor the code so instead of an instance, you deliver a **class**. Implement a **collection**, 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 function 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 it's result.

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

## 7. 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** all set trough the constructor and an array of **products** that is initially empty. 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  |
|---|
| <pre>let acc = new CheckingAccount('1314', 'ivan@some.com', 'Ivan', 'Petrov')</pre> |
| Output  |
| <pre>TypeError: Client ID must be a 6-digit number</pre>                            |

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

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

## Classes Interacting with DOM Elements

### 8. View Model

Do you remember the `sharedObject` problem from unit testing, we're going to build upon the idea and create an object that ensures a set of DOM elements and a JS object share the same state.

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:

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

All **\_elements** values and the **value** property should be linked. If the value of an element from **\_elements** changes 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

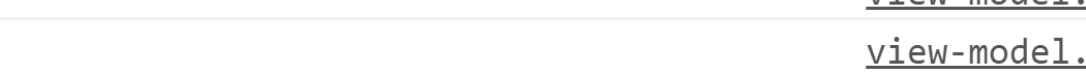
To help you test your code, you're provided with an **HTML** template:

| view-model.html   |
|---|
| <pre>&lt;!DOCTYPE html&gt; &lt;html lang="en"&gt; &lt;head&gt;   &lt;meta charset="UTF-8"&gt;   &lt;title&gt;Title&lt;/title&gt;   &lt;script src="https://code.jquery.com/jquery-3.1.1.min.js"&gt;&lt;/script&gt; &lt;/head&gt; &lt;body&gt; &lt;div id="wrapper"&gt;   &lt;input type="text" class="textbox"/&gt;   &lt;input type="text" class="textbox"/&gt; &lt;/div&gt; &lt;script src="view-model.js" defer&gt;&lt;/script&gt; &lt;/body&gt; &lt;/html&gt;</pre> |

And an example **JS** skeleton:

| view-model.js  |
|--|
| <pre>class Textbox {...}  let textbox = new Textbox(".textbox", /^[^a-zA-Z0-9]/); let inputs = \$(''.textbox');  inputs.on('input', function(){console.log(textbox.value)});</pre> |

Here is an example output in the browser:



The screenshot shows the Chrome DevTools Console with the 'Console' tab selected. There are three log entries, each with a message and a source location. The messages are 'a', 'as', and 'asd'. The source location for all three is 'view-model.js:35'. The console also shows a 'top' filter and a 'Preserve log' checkbox.

| Message | Source           |
|---------|------------------|
| a       | view-model.js:35 |
| as      | view-model.js:35 |
| asd     | view-model.js:35 |

The screenshot shows the Chrome DevTools Console with the 'Console' tab selected. The output of the `textbox.isValid()` method is displayed as `true` in green text. Above the console, a text input field contains the string 'asd'.

## Hints

- ## 9. \*Custom Form

The **Form** class should follow these requirements:

- 
- SOFTWARE UNIVERSITY  
FOUNDATION



- In case all passed objects are correct, the **Form** should **append** each of them to its **\_element** - in order of receiving them.
- The **Form** should have a private property **\_textboxes** containing all textboxes passed in.
- The **Form** should have a **submit()** method, when the **submit** method is called the following should happen:
  - The method should set all valid Textboxes (textboxes whose **isValid** method returns **true**) borders to **"2px solid green"** while setting all invalid Textboxes borders to **"2px solid red"**.
  - If all Textboxes are valid - returns **true**, otherwise returns **false**.
- The **Form** should have an **attach(selector)** method which attaches it to a passed in **selector**.

## 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:

| form.html  |
|--|
| <pre> &lt;!DOCTYPE html&gt; &lt;html lang="en"&gt; &lt;head&gt;   &lt;meta charset="UTF-8"&gt;   &lt;title&gt;Form&lt;/title&gt;   &lt;script src="https://code.jquery.com/jquery-3.1.1.min.js"&gt;&lt;/script&gt; &lt;/head&gt; &lt;body&gt; &lt;div id="wrapper"&gt;   &lt;span&gt;I am Wrapper&lt;/span&gt;   &lt;input type="text" id="username"/&gt;   &lt;input type="text" id="password"/&gt; &lt;/div&gt; &lt;div id="root"&gt;   &lt;span&gt;I am Root&lt;/span&gt; &lt;/div&gt; &lt;script src="form.js" defer&gt;&lt;/script&gt; &lt;/body&gt; &lt;/html&gt; </pre> |

And an example **JS** skeleton:

| form.js   |
|---|
| <pre> let result = (function() {    class Textbox {...}    class Form {...}    return {     Textbox: Textbox,     Form: Form   } })(); </pre> |

```

let Textbox = result.Textbox;
let Form = result.Form;
let username = new Textbox("#username", /^[^a-zA-Z0-9]/);
let password = new Textbox("#password", /^[^a-zA-Z]/);
username.value = "username";
password.value = "password2";
let form = new Form(username, password);
form.attach("#root");

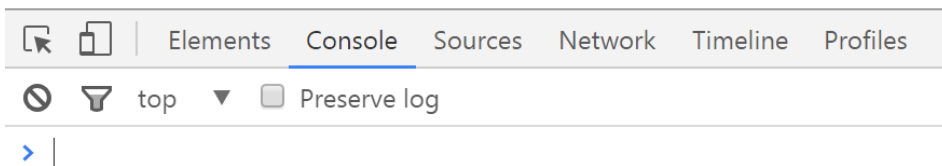
```

And here's an example in the browser:

I am Wrapper

I am Root

|          |           |
|----------|-----------|
| username | password2 |
|----------|-----------|

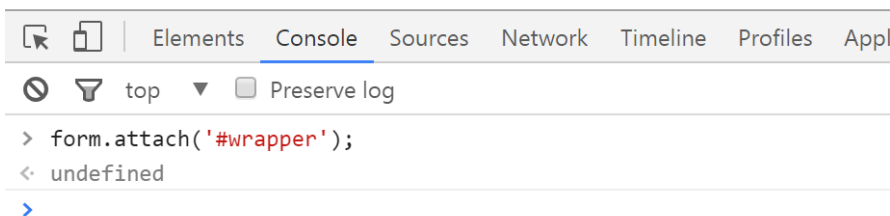


And after using **Form.attach()**

I am Wrapper

|          |           |
|----------|-----------|
| username | password2 |
|----------|-----------|

I am Root



You need to submit **ONLY the IIFE** (without the **"let result ="**) which returns an object with two properties **Textbox** and **Form** representing the above mentioned classes.