# Exercise: Classes & Unit Testing on Classes

Problems for exercises and homework for the ["JavaScript Advanced" course @ SoftUni](https://softuni.bg/trainings/3588/js-advanced-january-2022). Submit your solutions in the SoftUni judge system at <https://judge.softuni.bg/Contests/2769/Classes-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()); *// Tes...* ***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.