Exercise: Classes

1. Data Class

Write a **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`**.

Constraints

- The constructor of your class will receive **valid parameters**.
- Submit the class definition as is, **without** wrapping it in any function.

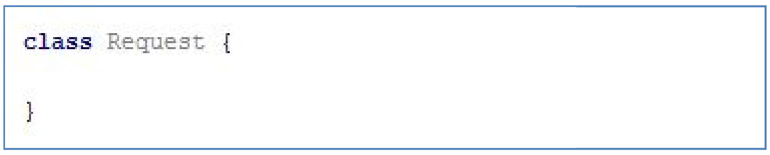
Examples

Sample Input	Resulting object

`let myData = new Request('GET', 'http://google.com', 'HTTP/1.1', '')`	`{ method: 'GET',`
	`uri: 'http://google.com',`
	`version: 'HTTP/1.1',`
	`message: '',`
	`response: undefined,`
	`fulfilled: false }`

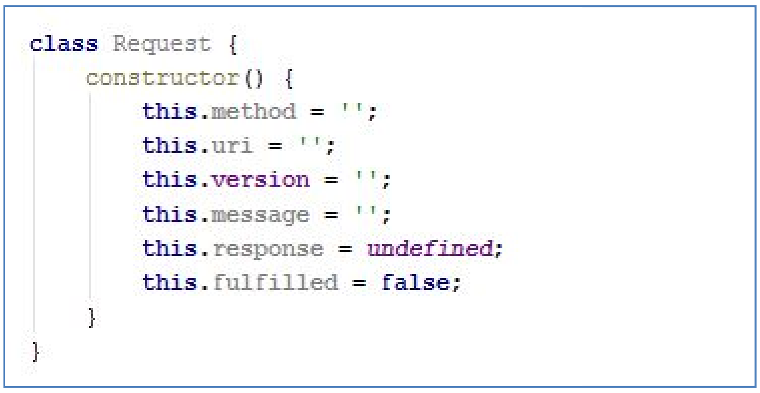
Hints

Using ES6 syntax, a class can be defined similar to a function, using the **`class`** keyword:



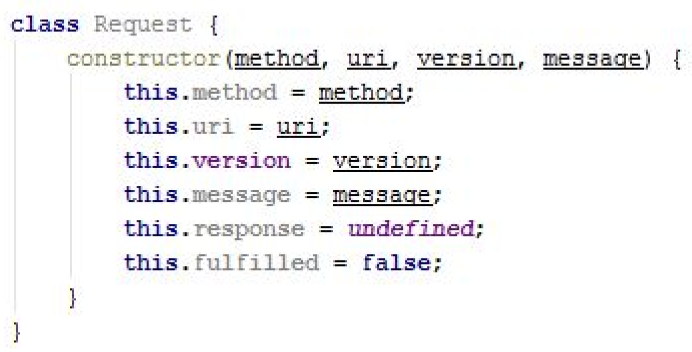
```
class Request {

}
```

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 the **`this`** identifier:



```
class Request {
    constructor() {
        this.method = '';
        this.uri = '';
        this.version = '';
        this.message = '';
        this.response = undefined;
        this.fulfilled = false;
    }
}
```

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:



```
class Request {
    constructor(method, uri, version, message) {
        this.method = method;
        this.uri = uri;
        this.version = version;
        this.message = message;
        this.response = undefined;
        this.fulfilled = false;
    }
}
```

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

What to submit?

Class Signature: class **Request**

2. 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 (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

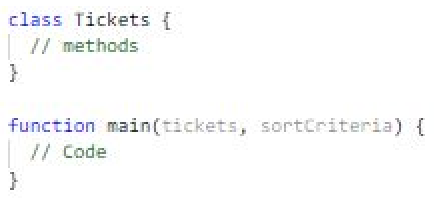
Sample Input	Output Array

`['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' } ]`

What to submit?

Class **Ticket** and a function **main**.



```
class Tickets {
  // methods
}

function main(tickets, sortCriteria) {
  // 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 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 it's 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
```

What to submit?

You are only required to submit the **Rat class**. No need to include the codes from the example above.

Class Signature: class **Rat**

4. Length Limit

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

LENGTH-LIMIT.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.

What to submit?

You are only required to submit the **Stringer class**. No need to include the codes from the example above.

Class Signature: class **Stringer**

5. Sorted List

Implement a **class**, 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.

Sample Input	Output

`let list = new List();`
`list.add(5);`
`list.add(6);`
	6
	7

`list.add(7);` `console.log(list.get(1));` `list.remove(1);` `console.log(list.get(1));`	

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 it’s result.
Your **add** and **remove functions** should **return** a class **instance** with the required functionality as it’s result.

What to submit?

You are only required to submit the **List class**. No need to include the codes from the example above.

Class Signature: class **List**