1. JavaScript: Staff List

1. JavaScript: Staff List

The task is to create a class StaffList. The class will manage a collection of staff members, where each member is uniquely identified by a name. The class must have the following methods:

1. add(name, age):
   - Parameters string *name* and integer *age* are passed to this function.
   - If *age* is greater than 20, it adds the member with the given name to the collection.
   - Else if *age* is less than or equal to 20, it throws an Error with the message 'Staff member age must be greater than 20'.
   - It is guaranteed that at any time, if a member is in the collection, then no other member with the same name will be added to the collection.
2. remove(name):
   - If the member with the given name is in the collection, it removes the member from the collection and returns true.
   - Else if the member with the given name is not in the collection, it does nothing and returns false.
3. getSize():
   - returns the number of members in the collection.

Your implementation of the class will be tested by a stubbed code on several input files. Each input file contains parameters for the functions calls. The functions will be called with those parameters, and the result of their executions will be printed to the standard output by the provided code. The stubbed code prints values returned by the *remove(name)* and *getSize()* functions, and it also prints messages of all the cached errors.

▼ Input Format For Custom Testing

The first line contains an integer, *n*, denoting the number of operations to be performed.

Each line *i* of the *n* subsequent lines (where $0 \leq i < n$) contains space-separated strings, such that the first of them is the function name and the remaining ones, if any, are parameters for that function.

▼ Sample Case 0

Sample Input For Custom Testing

```
5
add John 25
add Robin 23
getSize
remove Robin
getSize
```

Sample Output

```
2
true
1
```

Explanation

There are 2 staff members, 'John' and 'Robin', who are added by calling the *add* function twice. *getSize* is then called and returns the number of members in the collection, which is 2. Then, the staff member 'Robin' is removed from the list by calling the *remove* function, and since the given name is in the collection, the return value true is printed. Finally, *getSize* is called, which prints the size of the collect, which is now 1 because 'Robin' was removed.

▼ Sample Case 1

Sample Input For Custom Testing

```
5
add John 20
add Robin 10
getSize
remove Robin
getSize
```

Language: JavaScript (Node.js) Environment — Autocomplete Ready

```
'use strict';

class StaffList {
    //add your code here
    staffList = [];
    add(name, age){
        if(Number(age) > 20){
            if(this.staffList.find(i => i.name === name) != 0){
                this.staffList.push({name})
            }
            else{
                throw new Error("Staff member age must be greater than 20")
            }
        }
        else{
            throw new Error("Staff member age must be greater than 20")
        }
    }

    remove(name){
        const foundName = this.staffList.find(i => i.name == name);
        if(foundName){
            this.staffList = this.staffList.filter(i => i.name !== name);
            return true;
        }
        return false;
    }

    getSize(){
        return this.staffList.length;
    }
}

function main() {
```

Line: 55 Col: 18

Test Results | Custom Input | Run Code | Run Tests | Submit

Language: JavaScript (Node.js) Environment — Autocomplete Ready

```
'use strict';

class StaffList {
    //add your code here
    staffList = [];
    add(name, age){
        if(Number(age) > 20){
            if(this.staffList.find(i => i.name === name) != 0){
                this.staffList.push({name})
            }
            else{
                throw new Error("Staff member age must be greater than 20")
            }
        }
        else{
            throw new Error("Staff member age must be greater than 20")
        }
    }

    remove(name){
        const foundName = this.staffList.find(i => i.name == name);
        if(foundName){
            this.staffList = this.staffList.filter(i => i.name !== name);
            return true;
        }
        return false;
    }

    getSize(){
        return this.staffList.length;
    }
}

function main() {
```

Line: 55 Col: 18

Test Results | Custom Input | Run Code | Run Tests | Submit

```
getSize
remove Robin
getSize
```

Sample Output

```
2
true
1
```

Explanation
There are 2 staff members, 'John' and 'Robin', who are added by calling the *add* function twice. *getSize* is then called and returns the number of members in the collection, which is 2. Then, the staff member 'Robin' is removed from the list by calling the *remove* function, and since the given name is in the collection, the return value true is printed. Finally, *getSize* is called, which prints the size of the collect, which is now 1 because 'Robin' was removed.

▾ Sample Case 1

Sample Input For Custom Testing

```
5
add John 20
add Robin 10
getSize
remove Robin
getSize
```

Sample Output

```
Error: Staff member age must be greater than 20
Error: Staff member age must be greater than 20
0
false
0
```

Explanation
The function *add* is called to add 'John' and 'Robin', but in both cases, an error is thrown since *age* <= 20. Then, the *getSize* function is called and returns 0, which is the size of the collection, which is then printed. Next, the *remove* function is called to remove 'Robin', but since this staff member does not exist, false is returned and printed. Finally, the *getSize* function is called, which again returns 0, which is the size of the collection.

Language: JavaScript (Node.js) Environment — Autocomplete Ready

```
    remove(name){
        const foundName = this.staffList.find(i => i.name == name);
        if(foundName){
            this.staffList = this.staffList.filter(i => i.name !== name);
            return true;
        }
        return false;
    }

    getSize(){
        return this.staffList.length;
    }
}

function main() {
```

Line: 55 Col: 18

Test Results | Custom Input | Run Code | Run Tests | Submit

Compiled successfully. All available test cases passed

Test case 0 | Test case 1 | Test case 2 | Test case 3 | Test case 4 | Test case 5 | Test case 6

Input (stdin) — Run as Custom Input | Download

```
5
add John 25
add Robin 23
getSize
remove Robin
getSize
```

Your Output (stdout)

```
2
true
1
```

2. JavaScript Order List Processing

2. JavaScript: Order List Processing

Implement a function *processOrderList* that:

- takes 3 arguments: an order list *orderList*, a number *orderId*, and a string *state* that is either "*Processing*" or "*Delivered*". *orderList* is an array of order objects. An order object has the following schema:

```
{
   id: Number,
   state: String
}
```

- updates the order list depending on the state and returns the updated list.
  - If the state is '*Processing*', it updates the object in the list having id as *orderId*, to have the *state* '*Processing*'.
  - If the state is '*Delivered*', it deletes the object from the list having the id of *orderId*.
- If there is no order with the given *orderId* then the function returns the list *orderList* unchanged.

NOTE: You can assume that initially, all orders in the list have a *state* of '*Received*'.

The implementation will be tested by a provided code stub and several input files with parameters. The function will be called and the result printed to the standard output by the provided stubbed code. The stubbed code first creates a list of orders in *Received* state. The status of all orders in the updated order list are printed after all the operations are completed. If the final list is empty, the stub code prints 'All orders are in Delivered state'.

▾ Input Format For Custom Testing

The first line contains an integer, *n*, the number of orders in the list.
The second line contains an integer, *m*, the *numberOfOperations*.

Language: JavaScript (Node.js) Environment — Autocomplete Ready

```
'use strict';
function processOrderList(orderList, orderId, state) {
    if(state == "Delivered"){
        orderList = orderList.filter(a => a.id !== orderId)
    }
    else{
        orderList = orderList.map((a) =>{if (a.id == orderId) {a.state = "Processing"}
        return a});
    }
    return orderList;
}
```

Line: 22 Col: 1

Test Results | Custom Input | Run Code | Run Tests | Submit

Compiled successfully. All available test cases passed

Test case 0 | Test case 1 | Test case 2 | Test case 3 | Test case 4 | Test case 5 | Test case 6

Input (stdin) — Run as Custom Input | Download

```
3
2
1 Processing
2 Delivered
```

Your Output (stdout)

```
Order with id 1 is in Processing state
Order with id 3 is in Received state
```

Expected Output — Download

```
Order with id 1 is in Processing state
```

▾ Input Format For Custom Testing

New tab

The first line contains an integer, *n*, the number of orders in the list.
The second line contains an integer, *m*, the *numberOfOperations*.
Each line *i* of the *m* subsequent lines (where $0 \leq i < m$) contains 2 space-separated values. The first value is the *orderId* and the second is the updated order *state*.

▾ Sample Case 0

Sample Input For Custom Testing

```
STDIN              Function
-----              --------
3              →   n = 3
2              →   m = 2
1 Processing   →   orderId = 1 state =
'Processing'
2 Delivered    →   orderId = 2 state =
'Delivered'
```

Sample Output

```
Order with id 1 is in Processing state
Order with id 3 is in Received state
```

Explanation

Initially, the order list has 3 orders. The *state* of order 1 is updated to '*Processing*' and the *state* of order 2 is updated to '*Delivered*' and order 2 is deleted from the list. This makes the list contain the 1st and 3rd orders.

▾ Sample Case 1

Sample Input For Custom Testing

```
1
2
1 Processing
1 Delivered
```

Sample Output

```
All orders are in Delivered state
```

Explanation

Initially, the order list has 1 order. The *state* of order 1 is updated to '*Processing*' and then to '*Delivered*' at which point it is deleted from the list. This makes the list empty leading to all orders in the '*Delivered*' state.

Language: JavaScript (Node.js) Environment — Autocomplete Ready

```
'use strict';
function processOrderList(orderList, orderId, state) {
    if(state == "Delivered"){
        orderList = orderList.filter(a => a.id !== orderId)
    }
    else{
        orderList = orderList.map((a) =>{if (a.id == orderId) {a.state = "Processing"}
        return a});
    }
    return orderList;
}
```

Line: 22 Col: 1

Test Results | Custom Input | Run Code | Run Tests | Submit

Compiled successfully. All available test cases passed

Test case 0
Test case 1
Test case 2
Test case 3
Test case 4
Test case 5
Test case 6

Input (stdin) — Run as Custom Input | Download

```
1
2
1 Processing
1 Delivered
```

Your Output (stdout)

```
All orders are in Delivered state
```

Expected Output — Download

```
All orders are in Delivered state
```

2. JavaScript: Order List Processing

Implement a function *processOrderList* that:

- takes 3 arguments: an order list *orderList*, a number *orderId*, and a string *state* that is either "*Processing*" or "*Delivered*". *orderList* is an array of order objects. An order object has the following schema:

```
{
   id: Number,
   state: String
}
```

- updates the order list depending on the state and returns the updated list.
  - If the state is '*Processing*', it updates the object in the list having id as *orderId*, to have the *state* '*Processing*'.
  - If the state is '*Delivered*', it deletes the object from the list having the id of *orderId*.
- If there is no order with the given *orderId* then the function returns the list *orderList* unchanged.

NOTE: You can assume that initially, all orders in the list have a *state* of '*Received*'.

The implementation will be tested by a provided code stub and several input files with parameters. The function will be called and the result printed to the standard output by the provided stubbed code. The stubbed code first creates a list of orders in *Received* state. The status of all orders in the updated order list are printed after all the operations are completed. If the final list is empty, the stub code prints 'All orders are in Delivered state'.

▾ Input Format For Custom Testing

The first line contains an integer, *n*, the number of orders in the list.
The second line contains an integer, *m*, the *numberOfOperations*.

Language: JavaScript (Node.js) Environment — Autocomplete Ready

```
'use strict';
function processOrderList(orderList, orderId, state) {
    if(state == "Delivered"){
        orderList = orderList.filter(a => a.id !== orderId)
    }
    else{
        orderList = orderList.map((a) =>{if (a.id == orderId) {a.state = "Processing"}
        return a});
    }
    return orderList;
}
```

Line: 22 Col: 1

Test Results | Custom Input | Run Code | Run Tests | Submit

Compiled successfully. All available test cases passed

Test case 0
Test case 1
Test case 2
Test case 3
Test case 4
Test case 5
Test case 6

Input (stdin) — Run as Custom Input | Download

```
1
2
1 Processing
1 Delivered
```

Your Output (stdout)

```
All orders are in Delivered state
```

Expected Output — Download

```
All orders are in Delivered state
```