
# Given the function `bowdlerize(input, dictionary)` where:
- `input` is a string (e.g. "This is a cat")
- `dictionary` is a vector containing strings.

# Complete the following tasks:
- `input` should be of type `string`. If another type is given an `Error` is thrown with the message `Input should be a string`; (0.5 pts)
- `dictionary` is an array of `string`. If at least an element is not a `string` an `Error` is thrown with the message `Invalid dictionary format`; (0.5 pts)
- If `dictionary` contains words, they will be replaced in `input` with the first letter followed by a series of `*` characters followed by the last letter. The length of the resulting word will be the same as the original (e.g. 'test' will become 't**t'); (0.5 pts)
- A new string wil be returned, with `input` remaining unmodified; (0.5 pts)
- The function also returns the correct result for words starting with a capital letter. (0.5 pts)

```
function bowdlerize(input, dictionary)
{
        if (typeof input != "string") {
          throw new Error("Input should be a string");
        }
        dictionary.forEach(word => {
          if (typeof word != "string") {
            throw new Error("Invalid dictionary format");
          }
        });

        let sentence = input.split(" ");

        let censored = input;

        dictionary.forEach(wordD => {
          sentence.forEach(wordS => {
              let lowercase = wordS.toLowerCase();
            if (wordD === wordS || wordD === lowercase) {

              let newWord = wordS[0]+"*".repeat(wordS.length-2)+wordS[wordS.length-
1];

              censored = censored.replace(wordS,newWord);
            }
          });
        });
        return censored;
}


const app = {
    bowdlerize
};

module.exports = app;
```

# Given the function `removeOrderItem(orderInfo, position)` where:

```
- `orderInfo` is an object with the properties total and items
- `position` is an integer that determines one element in items

# Completati urmatoarele taskuri:
- validate `items` to be an `array`. If another type is given an error is thrown
with the message `Items should be an array`; (0.5 pts)
- Each object from `items` should have the properties price and quantity. If at
least one element is malformed an error is thrown with the message `Malformed
item`; (0.5 pts)
- `position` is validated in relation with the items array (0.5 pts)
- the function will return orderInfo without the element on the given position (0.5
pts)
- the total will be updated with the new order content. (0.5 pts)
function removeOrderItem(orderInfo, position){

    if(!Array.isArray(orderInfo.items)){
        throw new Error('Items should be an array')
    }

    for(let el of orderInfo.items){
        if(!el.price||!el.quantity){
        throw new Error('Malformed item')
        }
    }

    if(position<0||position>=orderInfo.items.length){
        throw new Error('Invalid position')
    }
    orderInfo.total-=
orderInfo.items[position].price*orderInfo.items[position].quantity
    orderInfo.items.splice(position,1)
    return orderInfo
}

const app = {
    removeOrderItem
};

module.exports = app;




--------------------------------------------------------------------
# Having the `function applyDiscount(vehicles, discount)`, complete the following
tasks: ---MERG 4

- Function should return a Promise; (0.5 pts)
- If `discount` is not a number, the function should `reject` an `Error` with the
message `Invalid discount`; (0.5 pts)
- `vehicles` is an array that contains objects with the following format: `{make:
string, price: number}` (Example: [{make: "Audi A5", price: 15000}]). If an array
with invalid objects is passed then the function should `reject` an `Error` with
the message `Invalid array format`; (0.5 pts)
- Function should `reject` a `string` with the value `Discount too big` if
`discount` is greater than 50% of the min price from `vehicles` array; (0.5 pts)
- Function should `resolve` an array with applied discount to each `vehicle price`;
(0.5 pts)

function applyDiscount(vehicles, discount) {
    return new Promise((resolve, reject) => {
      if (typeof discount !== 'number') {
        return reject(new Error('Invalid discount'));
      }

      if (!Array.isArray(vehicles) || vehicles.some(v => typeof v.make !== 'string'
|| typeof v.price !== 'number')) {
        return reject(new Error('Invalid array format'));
      }

      const minPrice = Math.min(...vehicles.map(v => v.price));
      if (discount > 0.5 * minPrice) {
        return reject('Discount too big');
      }

      resolve(vehicles.map(v => ({
        make: v.make,
        price: v.price * (1 - discount)
      })));
    });
  }
const app = {
    applyDiscount: applyDiscount
};

module.exports = app;
```

# Given the function `calculateFrequencies(input, stopWords)` where:
- `input` is a string (e.g. "This is an orange cat")
- `stopWords` is a vector containing strings.

# Complete the following tasks:
- `input` should be of type `string` or `String`. If another type is given an `Error` is thrown with the message `Input should be a string or a String`; (0.5 pts)
- `dictionary` is an array of `string` or `String`. If at least an element is not a `string` an `Error` is thrown with the message `Invalid dictionary format`; (0.5 pts)
- the function will calculate the relative frequencies of words in input and return a dictionary containing words as keys and frequencies as values (e.g. for the string 'orange cat' the result will be {orange : 0.5, cat : 0.5}); (0.5 pts)
- if stopWords contains any words, they will be ignored in the result (e.g. for the string 'the orange cat' with 'the' defined as a stopword the result will be {orange : 0.5, cat : 0.5}); (0.5 pts)
- the function also returns the correct result for words starting with a capital letter, which are considered identical to their lowercase variant. (0.5 pts)

```
function calculateFrequencies(input, dictionary) {
    if (typeof input !== 'string' && typeof input !== 'String') {
        throw "Input should be a string"
    }

    dictionary.forEach(element => {
        if (typeof element !== 'string' && typeof element !== 'String') {
            throw "Invalid dictionary format"
        }
    })


    //despart inputul (text) in cuvinte
    input = input.split(' ')
    dictionary.forEach(word => {
        input.forEach(element => {
            if (element.toLowerCase() === word.toLowerCase()) {
                //sterge elementul din input
                input.splice(input.indexOf(element), 1)
            }
        })
    })

    let obj = {}

    input.forEach(word => {
        if (!obj.hasOwnProperty(word.toLowerCase())) {
            obj[word.toLowerCase()] = 1
        }
        else {
            obj[word.toLowerCase()]++
        }
    })

    for (pr in obj) {
        obj[pr] = obj[pr] / input.length
    }
    return obj

}

const app = {
    calculateFrequencies
};

module.exports = app;

----------------------------------------------------------------------
# Given the classes `Duck` and `RubberDuck` where:
- a `Duck` is constructed based on a string name
- a `Duck` can swim
- a `RubberDuck` can float
- a `RubberDuck` can't swim

# Complete the following tasks:
- `name` should be of type `string` or `String`. If another type is given an
`Error` is thrown with the message `name must be string or String`; (0.5 pts)
- if a duck is instructed to swim it will return a string saying it is swimming
(e.g. if the duck is named `Donald` then the returned string is `Donald is
swimming`); (0.5 pts)
- a rubber duck is both a `Duck` and a `RubberDuck`; (0.5 pts)
- if a rubber duck is instructed to float it will return a string saying it floats
(e.g. if the rubber duck is named `Donald` then the returned string is `Donald
floats`); (0.5 pts)
- if a rubber duck is instructed to swim it will return a string saying cannot swim
(e.g. if the rubber duck is named `Donald` then the returned string is `Donald
```

can't swim, only float`); (0.5 pts)

```
class Duck {
    constructor(name){
        if(typeof name!=='string'){
            throw 'name must be string or String'
        }

        this.name = name

    }

    move(){
        return `${this.name} is moving`
    }
    swim(){
        return `${this.name} is swimming`
    }
}

class RubberDuck extends Duck{
    constructor(name){
        super(name)
    }

    float(){
        return `${this.name} floats`
    }
    swim() {
        return `${this.name} can't swim, only float`
    }
}

const app = {
    Duck,
    RubberDuck
}

module.exports = app
```

# Having the class `Queue` from file `index.js` implement the following tasks:
- Class `Queue` should contain a property called `items`, of type `Array` that will be initialized with an empty array (0.5 pts);
- Implement method `insert` that accepts `element` as an argument, which will be added in the array, according to the queue's principle;
- The method `insert` will allow only `string` elements to be added into the queue and will throw an Error with the text `Invalid Type` for other types.
- Implement method `extract` that will return an `element` from the array, according to the queue's principle;
- If the array is empty and the `extract` method is called, it will throw an Error with the text `Invalid Operation`;

```
class Queue {
    constructor(){
        this.items = []
    }
    insert (element){
        if (typeof element !== 'string'){
            throw ('Invalid Type')
        }
        this.items.push(element)
    }

    extract(element){
        if (this.items.length === 0){
            throw('Invalid Operation')
        }
        return this.items.shift()
    }
}

module.exports = Queue;
```

```
-------------------------------------------------------------------
# Given the function  `removeOrderItem(orderInfo, position)` where:
- `orderInfo` is an object with the properties total and items
- `position` is an integer that determines one element in items

# Completati urmatoarele taskuri:
- validate `items` to be an `array`. If another type is given an error is thrown
with the message `Items should be an array`; (0.5 pts)
- Each object from `items` should have the properties price and quantity. If at
least one element is malformed an error is thrown with the message `Malformed
item`; (0.5 pts)
- `position` is validated in relation with the items array (0.5 pts)
- the function will return orderInfo without the element on the given position (0.5
pts)
- the total will be updated with the new order content. (0.5 pts)


function removeOrderItem(orderInfo, position){
    if(!Array.isArray(orderInfo.items)){
        throw 'Items should be an array';
    }

    orderInfo.items.forEach(element=>{
    if(!(element.hasOwnProperty('price'))||!(element.hasOwnProperty('quantity')))
    {
        throw 'Malformed item';
    }

    });

    if(position>orderInfo.items.length){
        throw 'Invalid position';
    }

    let p=orderInfo.items[position].price;
    let q=orderInfo.items[position].quantity;

    orderInfo.items.splice(position,1);
    orderInfo.total=orderInfo.total-p*q;

    return orderInfo;

    }



    const app = {
        removeOrderItem
    };

    module.exports = app;
```

# Having the `function addTokens(input, tokens)` where:
- `input` is a string that could contain "..." for example: Subsemnatul ..., dominiciliat in ...;
- `tokens` is an array with token names .
- The function should replace each `...` from `input` with the values from `tokens` in the following format `${tokenName}`;

# Complete the following tasks:

- `input` should be a `string`. If other type is passed throw an `Error` with the message `Input should be a string`; (0.5 pts)
- `input` should be at least 6 characters long. If `input` length is less than 6 throw an `Error` with the message `Input should have at least 6 characters`; (0.5 pts)
- `tokens` is an array with elements with the following format: `{tokenName: string}`. If this format is not respected throw an `Error` with the following message `Invalid array format`; (0.5 pts)
- If `input` don't contain any `...` return the initial value of `input`; (0.5 pts)
- If `input` contains `...`, replace them with the specific values and return the new `input`; (0.5 pts)

```
function addTokens(input, tokens) {
  if (typeof input !== 'string') {
    throw new Error('Input should be a string');
  }
  if (input.length < 6) {
    throw new Error('Input should have at least 6 characters');
  }
  for (const token of tokens) {
    if (!token.tokenName || typeof token.tokenName !== 'string') {
      throw new Error('Invalid array format');
    }
  }
  if (!input.includes('...')) {
    return input;
  }
  let result = input;
  for (const token of tokens) {
    result = result.replace('...', `\${${token.tokenName}}`);
  }
  return result;
}
```

```
const app = {
    addTokens: addTokens
}

module.exports = app;
```

# Having the `function textProcessor(input, tokens)` where:
- `input` is a string that could contain tokens (Example: "Hello ${user}" or "Hello")
- `tokens` is an array with token names and values.
- All tokens are identified with the following format: `${tokenName}`

# Complete the following tasks:

- `input` should be a `string`. If other type is passed throw an `Error` with the message `Input should be a string`; (0.5 pts)
- `input` should be at least 6 characters long. If `input` length is less than 6 throw an `Error` with the message `Input should have at least 6 characters`; (0.5 pts)
- `tokens` is an array with elements with the following format: `{tokenName: string, tokenValue: string}`. If this format is not respected throw an `Error` with the following message `Invalid array format`; (0.5 pts)
- If `input` don't contain any token return the initial value of `input`; (0.5 pts)
- If `input` contains tokens, replace them with the specific values and return the new `input`; (0.5 pts)

# Having the `function textProcessor(input, tokens)` where:
- `input` is a string that could contain tokens (Example: "Hello ${user}" or "Hello")
- `tokens` is an array with token names and values.
- All tokens are identified with the following format: `${tokenName}`

# Complete the following tasks:

- `input` should be a `string`. If other type is passed throw an `Error` with the message `Input should be a string`; (0.5 pts)
- `input` should be at least 6 characters long. If `input` length is less than 6 throw an `Error` with the message `Input should have at least 6 characters`; (0.5 pts)
- `tokens` is an array with elements with the following format: `{tokenName: string, tokenValue: string}`. If this format is not respected throw an `Error` with the following message `Invalid array format`; (0.5 pts)
- If `input` don't contain any token return the initial value of `input`; (0.5 pts)
- If `input` contains tokens, replace them with the specific values and return the new `input`; (0.5 pts)

```
/*
Define a Widget object type is defined
The decorate function adds to Widget a method called enhance which increases the
size of a widget with n
```

```
If the parameter is not a number an exception is thrown ("InvalidType")
The method also works on already declared Widgets
*/


class Widget {
	constructor(name, size) {
		this.name = name;
		this.size = size;
	}

	getDescription() {
		return `a ${this.name} of size ${this.size}`;
	}
}

function decorate() {
	Widget.prototype.enhance = function(n) {
		if (typeof n !== 'number') {
			throw new Error('InvalidType');
		}
		this.size += n;
	};
}


module.exports.decorate = decorate
module.exports.Widget = Widget

-----------------------------------------------------------------------------------
----------------------------------------------------------
# Having the `function applyBonus(employees, bonus)`, complete the following tasks:

- Function should return a Promise; (0.5 pts)
- If `bonus` is not a number, the function should `reject` an `Error` with the
message `Invalid bonus`; (0.5 pts)
- `employees` is an array that contains objects with the following format: `{name:
string, salary: number}` (Example: [{name: "John Doe", salary: 5000}]). If an array
with invalid objects is passed then the function should `reject` an `Error` with
the message `Invalid array format`; (0.5 pts)
- Function should `reject` a `string` with the value `Bonus too small` if `bonus`
is less than 10% of the max salary from `employees` array; (0.5 pts)
- Function should `resolve` an array with applied bonus to each `employee salary`;
(0.5 pts)

function applyBonus(employees, bonus) {
    return new Promise((resolve, reject) => {
      if (typeof bonus !== "number") {
        reject(new Error("Invalid bonus"));
      } else if (!Array.isArray(employees)) {
        reject(new Error("Invalid array format"));
      } else if (employees.some(e => !e.name || !e.salary || typeof e.salary !==
"number")) {
        reject(new Error("Invalid array format"));
      } else if (bonus < Math.max.apply(Math, employees.map(e => e.salary)) * 0.1)
{
        reject("Bonus too small");
      } else {
        resolve(employees.map(e => ({ ...e, salary: e.salary + bonus })));
      }
    });
  }

let app = {
    applyBonus: applyBonus,
}

module.exports = app;
------------------------------------------------------------------------------------
--------------------------------------------------
/*
There is an object type called Bird
Define the Penguin type
A penguin is a child type for Bird and has an additional method called
swim(distance)
A penguin cannot be created without a name of type string
A penguin cannot fly and will say that if asked
A penguin can make a nest via its parent's method
See the tests for the accurate format of messages
*/

class Bird {
      constructor(name){
            this.name = name
      }

      fly(distance){
            return `${this.name} flies ${this.distance}`
      }

      makeNest(){
            return `${this.name} makes a nest`
      }
}

class Penguin extends Bird {
      constructor(name) {
      if (typeof name !== 'string') {
            throw exception('Penguin name must be a string');
      }
      super(name);
}
swim(distance) {
      return `${this.name} swims ${distance}`;
}

fly() {
      return `${this.name} is a penguin and cannot fly`;
}
}



module.exports.Bird = Bird
module.exports.Penguin = Penguin
```

```
--------------------------------------------------------------------------------------
----------------------------------
/*
 - the capitalize function receives as parameters a string and an array
 - the dictionary (the array) contains a series of words
 - in the initial text the words are separated by space
 - each dictionary term has to appear capitalized in the result
 - the result is a new string without modifying the initial one
 - if the text is not string or the dictionary not an array of strings an exception
is thrown (message is TypeError)
*/





function capitalize(text, dictionary){
    if (typeof text !== 'string' || !Array.isArray(dictionary) || !
dictionary.every(word => typeof word === 'string')){
        throw new TypeError('TypeError');
    }

    const words = text.split(' ');
    const capitalized = words.map(word => {
        for (let i = 0; i < dictionary.length; i++){
            if (word === dictionary[i].toLowerCase()){
                return word[0].toUpperCase() + word.slice(1);
            }
        }
        return word;
    });

    return capitalized.join(' ');
}


module.exports.capitalize = capitalize



--------------------------------------------------------------------------------------
--------------------------------------------


# Having the `function applyBlackFriday(products, discount)` where:
- `products` is an array of objects with the following format {name: string, price:
number};
- `discount` is a number that represents the percentage of discount to apply to the
products price.
- The function should return an array with applied discount to each product

# Complete the following tasks:

- Function should return a promise; (0.5 pts)
- `discount` should be a number, otherwise `reject` the promise with an `Error`
with the message `Invalid discount`; (0.5 pts)
- `discount` should be greater than 0 and less equals than 10, otherwise `reject`
the promise with an `Error` with the message `Discount not applicable`; (0.5 pts)
- `products` should contain elements in the specified format, otherwise `reject` an
```

`Error` with the message `Invalid array format`; (0.5 pts)
- Function should return the new array with the discounted products prices; (0.5 pts)

```
function applyBlackFriday(products, discount) {
    return new Promise((resolve, reject) => {
      if (typeof discount !== 'number') {
        return reject(new Error('Invalid discount'));
      }
      if (discount <= 0 || discount > 10) {
        return reject(new Error('Discount not applicable'));
      }
      if (!Array.isArray(products) || products.some(p => typeof p.name !== 'string'
|| typeof p.price !== 'number')) {
        return reject(new Error('Invalid array format'));
      }
      const discountedProducts = products.map(p => ({
        name: p.name,
        price: p.price - p.price * discount / 100
      }));
      resolve(discountedProducts);
    });
  }

const app = {
    applyBlackFriday: applyBlackFriday
};
module.exports = app;
```

# Having the `function processString(input)`, which initially splits the `input` string into `tokens` based on space, complete the following tasks:

- If any `token` is not a number, the function should throw an `Error`
- If `token` is not a number, the function should throw an `Error` with the message `Item is not a number`; (0.5 pts)
- If `input` is empty the function should return 100; (0.5 pts)
- Odd `tokens` are ignored; (0.5 pts)
- The function should return 100 minus the sum of all even `tokens`; (0.5 pts)

```
function processString(input) {
    if (!input) return 100;

    const tokens = input.split(" ");
    let sum = 0;

    for (let i = 0; i < tokens.length; i++) {
      const token = tokens[i];
      if (i % 2 === 0) {
        if (!Number.isFinite(+token)) {
          throw new Error("Item is not a number");
        }
        sum += +token;
      }
    }

    return 100 - sum;
  }

const app = {
    processString: processString
}

module.exports = app
```

# Having the `function toCamelCase(input)`, complete the following tasks:

- The function should throw an `Error` with the message `Input must be a string primitive or a string object` if input is number; (0.5 pts)
- The function should throw an `Error` with the message `Input must be a string primitive or a string object` if input is undefined; (0.5 pts)
- Given a `-` separated input, the function should return it in camelCase (with a lower first letter) (0.5 pts)
- Given a `_` separated input, the function should return it in camelCase (with a lower first letter) (0.5 pts)
- Given a `space` separated input, the function should return it in camelCase (with a lower first letter) (0.5 pts)

```
function toCamelCase(input) {
  if (typeof input !== "string") {
    throw new Error("Input must be a string primitive or a string object");
  }
  if (!input) {
    throw new Error("Input must be a string primitive or a string object");
  }

  let camelCased = "";
  for (let i = 0; i < input.length; i++) {
    if (input[i] === "-" || input[i] === "_" || input[i] === " ") {
      i++;
      camelCased += input[i].toUpperCase();
    } else {
      camelCased += input[i];
    }
  }
  return camelCased[0].toLowerCase() + camelCased.slice(1);
}

const app = {
  toCamelCase: toCamelCase
}

module.exports = app
```