# Exercises: Error Handling and Exceptions

Problems for exercises and homework for the [“JavaScript Advanced” course @ SoftUni](https://softuni.bg/courses/javascript-advanced). Submit your solutions in the SoftUni judge system at <https://judge.softuni.bg/Contests/346/>.

## 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 |
| let acc = new CheckingAccount('1314', 'ivan@some.com', 'Ivan', 'Petrov') |
| Output |
| TypeError: Client ID must be a 6-digit number |

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

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

## Request Validator

Write a JS function that validates an HTTP request object. The object has the properties method, uri, version and message. Your function must receive the object as a parameter and verify that each property meets the following requirements:

* method – can be **GET**, **POST**, **DELETE** or **CONNECT**
* uri – must be a valid resource address or an asterisk (\*); a resource address is a combination of alphanumeric characters and periods; all letters are Latin; the URI **cannot** be an empty string
* version – can be **HTTP/0.9**, **HTTP/1.0**, **HTTP/1.1** or **HTTP/2.0** supplied as a string
* message – may contain **any number** of non-special characters; special characters are <, >, \, &, ', "

If a request is valid, return it unchanged. If any part fails the check, throw an **Error** with message "**Invalid request header: Invalid {Method/URI/Version/Message}**". Replace the part in curly braces with the relevant word. Note that some of the **properties may be missing**, in which case the request is invalid. Check the properties in the order in which they are listed here. If more than one property is invalid, throw an error for the first encountered.

### Input / Output

Your function will receive an object as a parameter. As output, **return** the same object or throw an **Error** as described above.

### Examples

|  |  |
| --- | --- |
| Sample Input | Output |
| validateRequest({  method: 'GET',  uri: 'svn.public.catalog',  version: 'HTTP/1.1',  message: ''  }); | {  method: 'GET',  uri: 'svn.public.catalog',  version: 'HTTP/1.1',  message: ''  } |
| Sample Input | Output |
| validateRequest({  method: 'OPTIONS',  uri: 'git.master',  version: 'HTTP/1.1',  message: '-recursive'  }); | Invalid request header: Invalid Method |

|  |  |
| --- | --- |
| Sample Input | Output |
| validateRequest({  method: 'POST',  uri: 'home.bash',  message: 'rm -rf /\*'  }); | Invalid request header: Invalid Version |

### Hints

Since validating some of the fields may require the use of **RegExp**, you can check your expressions using the following samples:

|  |  |
| --- | --- |
| URI | |
| Valid | Invalid |
| svn.public.catalog  git.master  version1.0  for..of  .babelrc  c | %appdata%  apt-get    home$  define apps  "documents" |

* Note that the URI **cannot** be an empty string.

|  |  |
| --- | --- |
| Message | |
| Valid | Invalid |
| -recursive  rm -rf /\*  hello world  https://svn.myservice.com/downloads/  %root% | <script>alert("xss vulnerable")</script>  \r\n  &copy;  "value"  '; DROP TABLE |

* Note that the message **may** be an empty string, but the property must still be present.

## Parse Data

Create a JS program that defines a class Candy and a function that receives an array of recipes and generates class instances from them. The **Candy** has topping, filling and spice, all set trough the constructor as strings. The recipes will be an array of strings in format:

{topping}:{filling1},{filling2},{…}:{spice}

There can be any number of comma-separated **fillings** in the recipe and zero or one type of **spice**. If one of the parameters is not specified (it is empty string) set it to null. If this format is not followed, consider the recipe invalid. Each instance of **Candy** must have valid initial state, but not all recipes will be correct and the constructor must throw a TypeError if the following requirements aren’t met:

topping – must be either '**milk chocolate**', '**white chocolate**' or '**dark chocolate**'; any other topping, including null is invalid

filling – can be null or a combination of up to three of the following: **'hazelnut'**, **'caramel'**, **'strawberry'**, **'blueberry'**, **'yogurt'**, **'fudge'**; if more than 3 fillings are specified or the filling is not in the list, the recipe is invalid

spice – can be null or any string except **'poison'** or **'asbestos'**

As a result, return an array containing all valid instances – if an instance was created with invalid parameters, do not include it in the final array. Do not display any errors as you encounter them.

### Input / Output

Your function will receive an array of string as input. As output, **return** an array of valid objects. Do not display any errors.

Submit a function that receives one parameter and holds the class definition inside it. The properties of the class must be named exactly as described (either the data property, or the getter).

### Examples

|  |
| --- |
| Sample Input |
| makeCandy([  'milk chocolate:hazelnut,caramel:pumpkin',  'dark chocolate::chips',  'white chocolate::poison', *// invalid*  'white chocolate:fudge:',  'frosting:yogurt:frosting', *// invalid*  'dark chocolate:blueberry:rock crystals'  ]) |
| Output |
| [ Candy {  topping: 'milk chocolate',  filling: 'hazelnut,caramel',  spice: 'pumpkin' },  Candy {  topping: 'dark chocolate',  filling: null,  spice: 'chips' },  Candy {  topping: 'white chocolate',  filling: 'fudge',  spice: null },  Candy {  topping: 'dark chocolate',  filling: 'blueberry',  spice: 'rock crystals' } ] |

## Object Iterator

Create a function which takes in an object and returns an iterator for it. The iterator should have a method next() which returns an object containing properties value - holding the next element and done (true or false) - signifying if the iterator has finished. The iterator should enumerate all keys of the object **alphabetically in descending order**.

### Examples

|  |  |
| --- | --- |
| Sample Input | Output |
| **let *obj*** = {**age**: 27, **name**: **"pesho"**, **book**: **"Lord of the Rings"**};  **let *iterator*** = *makeIterable*(***obj***);  **while**(**true**){  **let *res*** = ***iterator***.next();  **if**(***res***.**done**) **break**;  **console**.log(***res***.**value**);  } | name  book  age |
| **let *obj*** = {**name**: **"gosho"**, **"13"**: **true**, **book**: **"Lord of the Drinks"**, 2: 2, **age**: 15, **passportNumber**: 12345678};  **let *iterator*** = *makeIterable*(***obj***);  **while**(**true**){  **let *res*** = ***iterator***.next();  **if**(***res***.**done**) **break**;  **console**.log(***res***.**value**);  } | passportNumber  name  book  age  2  13 |

### Template

Submit in the judge the JS code (implementation) of the makeIterable function. Here's an example template:

|  |
| --- |
| make-iterable.js |
| **function** *makeIterable*(object) {  *//****TODO***  **return** {  next: **function** () {  *//****TODO***  }  }  } |

### Hints

* The comparing should be done alphabetically so you can use localeCompare. Keep in mind that numbers also should be compared by alphabetically.

## Random Generator

Write a JS Generator function that yields pseudo-random numbers between 0 and 99 by supplied seed. When using the same seed to initialize two different generators, they should produce an identical sequence. Use the following formula to calculate the value of the next number:



Where ***xn*** is the next number in the sequence, ***xn-1*** is the previous number in the sequence, ***Mod*** is the modulo (remainder) operator and ***p*** = 4871 and ***q*** = 7919. When calculating the first element in the sequence, use the seed for the value of ***xn-1***. See the examples for detailed calculations.

### Input / Output

As input, your function will receive a seed value as a number. As output you need to **yield** the values as described.

### Examples

|  |  |  |
| --- | --- | --- |
| Sample Input | Output | Explanation |
| let rnd = random(100);  for (let i = 0; i < 10; i++) {  console.log(rnd.next().value);  } | 0  2  29  89  34  76  47  83  30  40 | When calculating the first number, we take the value of the seed for ***xn-1*** in the formula. For ***xn*** we get:  (100 \* 100) % (4871 \* 7919) = 10000  Since we need a number between 0 and 99, we further apply modulo 100 to this result and yield it:  10000 % 100 = 0  The next number starts with 10000 as the value of ***xn-1*** to arrive at:  100002 % 38573449 = 22853102 % 100 = 2 |

## Fibonacci Generator

Write a JS Generator function that yields numbers from the Fibonacci sequence. Assume the first two elements in the sequence are **1** and **1**.

### Input / Output

There will be no input. As output, your function needs to **yield** the values as described.

### Examples

|  |  |
| --- | --- |
| Sample Input | Output |
| let fib = fibonacci();  console.log(fib.next().value);  console.log(fib.next().value);  console.log(fib.next().value);  console.log(fib.next().value);  console.log(fib.next().value);  console.log(fib.next().value);  console.log(fib.next().value); | 1  1  2  3  5  8  13 |
| let fib = fibonacci();  for (let number of fib) {  console.log(number);  } | 1  1  2  3  5  … *// series is infinite* |

## \*\*Look and Say Generator

Have you heard of the Look and Say Sequence? It is an interesting sequence that works like this:  
**1** -> **11** -> **21** -> **1211** -> **111221** -> **312211**

You start off with a number in this instance **1** - it is the first element in the sequence - the next element **11** is calculated in the following way - you take the previous element (**1**) and you **compress all** **consecutive same digits** **into one** and in front of it you write the **number of digits compressed** the **blue** number here is the count of the digits compressed and the **red** number is the digit that was compressed. This can be more easily explained with the 5th and 6th elements - **111221** -> we have **3** consecutive **1**'s so we compress them into **31** (**3** times the digit **1**) then we have **2** consecutive **2**'s, we compress them into **22** (**2** times the digit **2**) and finally we have **1** times the digit **1** - we compress it into **11** (**1** time the digit **1**) thus we get **312211**.

Create a generator function which takes as a parameter a number and can produce the next elements from the sequence as a **STRING**.

### Examples

|  |  |
| --- | --- |
| Sample Input | Output |
| **let *lookSequence*** = *lookAndSay*(1);  **console**.log(***lookSequence***.next().**value**);  **console**.log(***lookSequence***.next().**value**);  **console**.log(***lookSequence***.next().**value**);  **console**.log(***lookSequence***.next().**value**);  **console**.log(***lookSequence***.next().**value**); | 11  21  1211  111221  312211 |
| **let *lookSequence*** = *lookAndSay*(113);  **console**.log(***lookSequence***.next().**value**);  **console**.log(***lookSequence***.next().**value**);  **console**.log(***lookSequence***.next().**value**);  **console**.log(***lookSequence***.next().**value**);  **console**.log(***lookSequence***.next().**value**); | 2113  122113  11222113  21322113  121113222113 |

### Template

Submit in the judge the JS code (implementation) of the lookAndSay function. Here's an example template:

|  |
| --- |
| make-iterable.js |
| **function** \* *lookAndSay*(start) {  *//****TODO***  **yield** nextElement;  } |

### Hints

* You can check more about the [Look and Say Sequence](https://en.wikipedia.org/wiki/Look-and-say_sequence) in Wikipedia.
* Think of a way to get all consecutive digits from a number, treating it as a string may be easier.