JavaScript Exams

Disclaimer: do not use any third-party library unless explicitly stated in the task!

## Chapter 1

### Exam 1.1

Write a JavaScript app that accepts an array of numbers and returns an integer indicating the count of odd elements in the array.

|  |  |
| --- | --- |
| Input | Output |
| [6, 4, 3, 1, 9, 44, 33, 2] | 4 |

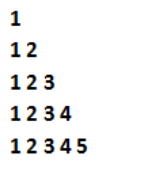
### Exam 1.2

Write a JavaScript app that performs a bubble sort on an array of numbers that is passed as an argument and the result is a new instance that is sorted. Do not use the native array.sort method!

|  |  |
| --- | --- |
| Input | Output |
| [6, 4, 3, 1, 9, 44, 33, 2] | [2, 3, 4, 6, 9, 33, 44] |

### Exam 1.3

Create JavaScript apps that print the following patterns. Note – one app per pattern and all apps must accept a parameter that dynamically defines the length of the numbers printed.

Pattern 1:  


#### Pattern 2:

#### Pattern 3:

#### Pattern 4:

#### Pattern 5:

#### Pattern 6:

#### Pattern 7:

### Exam 1.4

Create a JavaScript function that accepts array of elements of random type and it returns an object with the following structure:   
  
{  
 nullCount: 3,  
 objectsCount: 2, // objects that are not null or array  
 stringsCount: 5,  
 numericsCount: 0, // Number, float, BigInt etc.  
 arraysCount: 2,  
 undefinedCount: 6,  
 datesCount: 3,  
 booleansCount: 0,  
 functions: 2  
}

Which lists the count of the types of elements in the array that was passed as a parameter of the function.

|  |  |
| --- | --- |
| Input | Output |
| [6, "Test", "value", 1, undefined, null, () => {console.log("Hello, world!")}, {count: 5}] | {  nullCount: 1,  objectsCount: 2,  stringsCount: 2,  numericsCount: 2,  arraysCount: 0,  undefinedCount: 1,  datesCount: 0,  booleansCount: 0,  functions: 1  } |

Exam 1.5

Create a JavaScript function that calculates which of two objects passed as parameters is “heavier”.

#### Part 1

Iterate over each of the first-level properties of the objects and calculate the heaviness of the object by the following formula:

|  |  |
| --- | --- |
| Property Type | Value |
| Object, Array, Function | 10 |
| string | 8 |
| null | 2 |
| Numeric (Number, float, BigInt etc.) | 4 |
| date | 5 |
| undefined | 2 |
| Boolean | 4 |

|  |  |
| --- | --- |
| Input | Output |
| { cards: 6,  label: "Test",  description: "value",  type: 1,  role: undefined,  person: {name: "John Doe" },  skills: null,  report: () => {console.log("Hello, world!")},  experience: {count: 5}  } | 58 |

#### Part 2

Calculate the weight by iterating over child objects and estimate their weight as well by using the formulas in point a with the condition that an object is always worth 10 points, every child property adds to that amount.

|  |  |
| --- | --- |
| Input | Output |
| {  cards: 6,  label: "Test",  description: "value",  type: 1,  role: undefined,  person: {name: "John Doe" },  skills: null,  report: () => {console.log("Hello, world!")},  experience: {count: 5}  } | 70 |

### Exam 1.6

Create a JavaScript function that accepts two parameters - a date object and a string. The result of the function should be a formatted string representation of the date object. The string indicates the format the date object must be transformed into.

|  |  |
| --- | --- |
| Input | Output |
| myFunction(new Date(), "YYYY-MMM-DD HH:mm:ss Is my proof of concept!") | "2021-May-21 10:21:33 Is my proof of concept! " |

#### Part 1

The tokens passed that would indicate the format are:

|  |  |  |
| --- | --- | --- |
| Datetime part | Token | Output |
| Month | M | 1 2 ... 11 12 |
|  | MM | 01 02 ... 11 12 |
| Day of Month | D | 1 2 ... 30 31 |
|  | DD | 01 02 ... 30 31 |
| Year | YYYY | 1970 1971 ... 2029 2030 |
| AM/PM | A | AM PM |
|  | a | am pm |
| Hour | H | 0 1 ... 22 23 |
|  | HH | 00 01 ... 22 23 |
| Minute | m | 0 1 ... 58 59 |
|  | mm | 00 01 ... 58 59 |
| Second | s | 0 1 ... 58 59 |
|  | ss | 00 01 ... 58 59 |

Obviously, you are not allowed to use moment.js or any other date formatting library!!!

#### Part 2

Expand the table in Part 1 by adding the following tokens:

|  |  |  |
| --- | --- | --- |
| Datetime part | Token | Output |
| Month | MMM | Jan Feb ... Nov Dec |
|  | MMMM | January February ... November December |
| Quarter | Q | 1 2 3 4 |
| Day of Month | Do | 1st 2nd ... 30th 31st |
| Day of Week | ddd | Sun Mon ... Fri Sat |
|  | dddd | Sunday Monday ... Friday Saturday |
| Day of Week (ISO) | E | 1 2 ... 6 7 |
| Week of Year (ISO) | W | 1 2 ... 52 53 |
|  | Wo | 1st 2nd ... 52nd 53rd |
|  | WW | 01 02 ... 52 53 |
| Time Zone | z or zz | EST CST ... MST PST Note: as of **1.6.0**, the z/zz format tokens have been deprecated from plain moment objects. [Read more about it here.](https://github.com/moment/moment/issues/162) However, they \*do\* work if you are using a specific time zone with the moment-timezone addon. |

## Chapter 2

### Exam 2.1

Expand the Array method and implement the native array.sort method. The method accepts a function handler as a single parameter that defines how a and b are compared and the method extension returns the sorted array. The sorting must work with any data types in the array. Result should be the same array but sorted!  
  
Use of the native array.sort method is not allowed!

|  |  |
| --- | --- |
| Input | Output |
| sortArray([6, 4, 3, 1, 9, 44, 33, 2], (left, right) => { return left < right}) | [2, 3, 4, 6, 9, 33, 44] |

#### Part 1

Implement the bubble sorting algorithm

#### Part 2

Implement the min-max sorting algorithm

### Exam 2.2

Create a JavaScript function that accepts two parameters - an array and an object and:

#### Part 1

Returns true or false if this object is added in the array or not (compare reference).

|  |  |
| --- | --- |
| Input | Output |
| const person = {name: "John Doe" }    findObj([6, "Test", "value", person, 1, undefined, null, () => {console.log("Hello, world!")}, {count: 5}, {name: "John Doe" }], person) | true |

#### Part 2

Expand the function to perform a deep search as well (find the object in every property of each object recursively)

|  |  |
| --- | --- |
| Input | Output |
| const person = {name: "John Doe" }    findObj([6, "Test", "value", 1, undefined, null, () => {console.log("Hello, world!")}, {count: 5}, {name: "John Doe", person: person }], person) | true |

#### Part 3

Returns true or false if object with the same property values (non-reference types) exists in the array

|  |  |
| --- | --- |
| Input | Output |
| const person = {name: "John Doe" }    findObj([6, "Test", "value", 1, undefined, null, () => {console.log("Hello, world!")}, {count: 5}, {name: "John Doe" }], person) | true |

#### Part 4

Return true or false if object with the same properties exists in the array, both value types and reference types

Merge the solutions from Part 2 and 3.

### Exam 2.3

Create a JavaScript function that accepts two parameters – array and an array

#### Part 1

The function should append to every member of the array type of the first array only the elements in the second array that are “truthy”.

|  |  |
| --- | --- |
| Input | Output |
| const arr1 = [1, undefined, [1, 2, 3], "test", {name: "John Doe"}]  const arr2 = [null, () => {console.log("Hello, world!")}, ["one", "five"], undefined, 6]    appender(arr1, arr2) | [1, undefined,  [1, 2, 3,  () => {console.log("Hello, world!")},  ["one", "five"], undefined, 6],  "test", {name: "John Doe"}] |

#### Part 2

The function should append to the array elements of the first array the following elements from the second array – if the index of the array child in the first array is even, append only the values of the second array that are of Array type (unpacking them and appending their content), if the index is uneven – append the object elements from the second array.

|  |  |
| --- | --- |
| Input | Output |
| const arr1 = [[1, 2, 3, 4], ["one", "two"], [5, 6]]    const arr2 = [null, () => {console.log("Hello, world!")}, ["one", "five"], {role: "admin"}, {name: "John"}, [1000, 1001]]    appender(arr1, arr2) | [[1, 2, 3, 4, "one", "five", 1000, 1001], ["one", "two", {role: "admin"}, {name: "John"}], [5, 6, "one", "five", 1000, 1001]] |

### Exam 2.4

Create a JavaScript function that accepts two arguments - an array with numbers and a number and:

#### Part 1

Returns the sum of the elements in the array that have value higher than the number passed as argument

|  |  |
| --- | --- |
| Input | Output |
| const arr1 = [1,2,3,4,5,6,7]  const higherThan = 4  calculate(arr1, higherThan) | 18 |

#### Part 2

Returns a new array with the elements that are divisible by the number passed into the array

|  |  |
| --- | --- |
| Input | Output |
| const arr1 = [1,2,3,4,5,6,7]  const divisible = 2  calculate(arr1, divisible) | [2, 4, 6] |

### Exam 2.5

Create a JavaScript function that accepts an array as the argument and returns the same array with every object in the array flattened. This means, each array and object property of the members of the array is replaced by its elements.

|  |  |
| --- | --- |
| Input | Output |
| [{  person: {  firstName: "John",  lastName: "Doe",  role: "Admin"  },  permissions: ["read", "write", "special"],  age: 42,  competencies: [{skill: "JavaScript", level: "junior"}, {skill: "css", level: "junior"}]  }] | [{  person\_firstName: "John",  person\_lastName: "Doe",  person\_role: "Admin",  permissions\_0: "read",  permissions\_1: "write",  permissions\_2: "special",  age: 42,  competencies\_0\_skill: "JavaScript",  competencies\_0\_level: "junior",  competencies\_1\_skill: "css",  competencies\_1\_level: "junior"  }] |

### Exam 2.6

Create a JavaScript function that accepts an array and a number between 0 and 100 and removes the inner percentage of the elements from that array. Round up to the bigger number the number of elements to be removed so that there is an even amount left from both ends of the original array.

|  |  |
| --- | --- |
| Input | Output |
| const arr = [1,2,3,4,5,6,7,8,9,10]  const percentage = 50  removeInner(arr, percentage) | [1,2,9,10] |

### Exam 2.7

Create a random JavaScript function that takes an array of numbers, randomizes their position and returns an array with the longest sequence of elements that ascend in numeric order.

|  |  |
| --- | --- |
| Input | Output |
| const arr = [1,2,3,4,5,6,7,8,9,10]  randomize(arr)  // randomized as: [10, 3, 2, 5, 6, 9, 8, 1, 4, 7] | [2,5,6,9] |

### Exam 2.8

Create a JavaScript function that accepts two arguments - an array with elements and a numeric value – the weight.

#### Part 1

The function must return true or false if there is an element in the array that has the same weight (See Exam 1.5) as passed in the argument

|  |  |
| --- | --- |
| Input | Output |
| const arr = [6, "Test", "value", 1, undefined, null, {name: "john.doe", role: "admim"}]  const weight = 16    findByWeight(arr, weight) | true |

#### Part 2

Another function with similar parameters must return true or false if all elements in the array have a higher weight than passed argument

|  |  |
| --- | --- |
| Input | Output |
| const arr = [6, "Test", "value", 1, 5, {name: "john.doe", role: "admim"}]  const weight = 2    findByWeight(arr, weight) | true |

## Chapter 3

### Exam 3.1

Extend the Array class with the functionalities developed in the following tasks:

* Part 1 – Task 1
* Part 1 – Task 2
* Part 1 – Task 4
* Part 2 – Task 2
* Part 2 – Task 3
* Part 2 – Task 5
* Part 2 – Task 6

### Exam 3.2

Extend the Date class with the functionality developed in Exam 1.6

### Exam 3.3

Create a new JavaScript class called Color. It should have a constructor that takes three digits for RGB color, a string for hexadecimal representation and if no parameter passed – default to black. Implement the following methods

* 1. getColorRGB – returns the code in CSS RGB format
  2. getColorShortHex - returns the color in CSS Hex string format
  3. getColorLongHex - returns the color in CSS Hex string format
  4. Update the constructor to support passing opacity as well

### Exam 3.4

Create a JavaScript class that represents a classic watch. It should be instantiated with a constructor that takes a Date object. If the Date object is not passed, it takes the current time. The class should expand the following methods:

* 1. showTime() - when the function is called, it starts printing in the console the time every second in the default Date format.
  2. showTime(format) - extend the functionality by adding the format argument to the function which prints the data every second using the specified format string (See Part 1 – Task 6)
  3. showTime(format, timezone) - extends the functionality by adding the timezone parameter which prints the time adjusted for the timezone passed. The timezone argument is IANA type (<https://en.wikipedia.org/wiki/List_of_tz_database_time_zones>) see the TZ database name column
  4. Expand the showTime function so that every new invocation of the showTime function with different timezone shall print the time independently. For example, if I call showTime with European Central Time once and then again with Asia/Shanghai, then every second two numbers shall be printed, once for ECT and once for Asia/Shanghai. If I call again the function with already existing timezone, the watch Tshould stop printing the time for that particular timezone.
  5. Extend the showTime function so that it accepts a third parameter – a function callback that receives an array of strings and is called by the watch class every time the date updates and prints the output.
  6. hideTime(timezone) - this method cancels the time printing stream for a timezone
  7. chronoStart(), chronoStop() - starts a chronometer, when stop is called, prints the ellapsed time in seconds and milliseconds.

### Exam 3.5

Extend the built-in Object class by adding the deepCopy method which should accept two parameters – required first parameter is an object, and optional second parameter object. If a second parameter is passed, the method returns a new object with the properties of the second parameter object and with all the properties of the first parameter copied to the second, overwriting existing properties. If no second parameter is passed, the method returns a new object with all properties deeply copied from the first object.  
  
Definition of deep copy – a process of copying properties not only of the first level object properties but the nested properties. All reference types must be created anew by value. Functions must be copied as well.

|  |  |
| --- | --- |
| Input | Output |
| const obj = {  person: {  firstName: "John",  lastName: "Doe",  role: "Admin"  },  permissions: ["read", "write", "special"],  age: 42,  competencies: [{skill: "JavaScript", level: "junior"}, {skill: "css", level: "junior"}]  }    Object.deepCopy({}, obj) | {  person: {  firstName: "John",  lastName: "Doe",  role: "Admin"  },  permissions: ["read", "write", "special"],  age: 42,  competencies: [{skill: "JavaScript", level: "junior"}, {skill: "css", level: "junior"}]  } |

### Exam 3.6

Create a NodeJS script that accepts the following command line arguments:

|  |  |  |
| --- | --- | --- |
| Argument name | Value type | Rules |
| --path | String, path to a folder | Optional, must be valid and existing path if passed  Default: current location |
| --fileTypes | String, comma separated file type extensions | Optional, must be valid extensions in the .ts, .txt, .js type  Default: .js |
| --exclude | String, comma separated file and folder names that must be excluded | Optional, must be valid file and folder names if passed  Default: node\_modules |
| --help | No value needed | Optional |

Sample execution:

|  |
| --- |
| node lineCounter.js --fileTypes=.js,.ts,.json --exclude=node\_modules,tsconfig.ts,package.json,dist |

The NodeJS script should traverse all files and locations in the specified --path (Current location, by default) and count the lines of text in the specified by –fileTypes file types. The script shall print the total lines of code/text for each of the --fileTypes passed in the command line.  
  
*Optional: 1. Implement the --help flag that prints in console all the arguments, their accepted values and defaults.*

*Optional 2: Add a sym-link/shortcut check to prevent circular traversing of the file system.*

### Exam 3.7

(optional) Create a JavaScript function that implements a maze using the recursive backtracking algorithm (See [here)](https://medium.com/swlh/how-to-create-a-maze-with-javascript-36f3ad8eebc1)

#### Part 1

The maze should be n x m size and contain an entrance and an exit

#### Part 2

The function should display the resulting mase in an HTML file with the walls and paths visible

#### Part 3

Add a parameter to the function that displays the initial state of the maze (all paths or all walls) and visualizes the progression of the generation of the maze by adding every step of the generation every X milliseconds.

Example:  
//upload.wikimedia.org/wikipedia/commons/transcoded/b/b1/MAZE\_30x20\_Prim.ogv/MAZE\_30x20\_Prim.ogv.480p.vp9.webm

### Exam 3.8

Create a dungeon crawler function that uses the Maze generation in Exam 3.7 and updates it by:

#### Part 1

Add a configurable number of gems in the maze on random locations

#### Part 2

Find the most optimal way (with least steps taken) to collect all gems and exit

#### Part 3

Visualize the progression of the crawler in an HTML page (visualization is free to be decided) by showing the steps the crawler takes in their final progression

#### Part 4

Find the most optimal way to collect the greatest number of gems given a limit of walks that can be taken – for example, how many gems can the crawler get and exit the dungeon with only 50 steps in a 15 x 15 maze?