Working With Arrays

Coding Challenge #1

Julia and Kate are doing a study on dogs. So each of them asked 5 dog owners about their dog's age, and stored the data into an array (one array for each). For now, they are just interested in knowing whether a dog is an adult or a puppy.
A dog is an adult if it is at least 3 years old, and it's a puppy if it's less than 3 years old.

Your tasks:

Create a function `'checkDogs'`, which accepts 2 arrays of dog's ages (`'dogsJulia'` and `'dogsKate'`), and does the following things:

1. Julia found out that the owners of the **first** and the **last two** dogs actually have cats, not dogs! So create a shallow copy of Julia's array, and remove the cat ages from that copied array (because it's a bad practice to mutate function parameters)
2. Create an array with both Julia's (corrected) and Kate's data
3. For each remaining dog, log to the console whether it's an adult (*"Dog number 1 is an adult, and is 5 years old"*) or a puppy (*"Dog number 2 is still a puppy 🐶"*)
4. Run the function for both test datasets

Test data:

- Data 1: Julia's data `[3, 5, 2, 12, 7]`, Kate's data `[4, 1, 15, 8, 3]`
- Data 2: Julia's data `[9, 16, 6, 8, 3]`, Kate's data `[10, 5, 6, 1, 4]`

Hints: Use tools from all lectures in this section so far 😉

GOOD LUCK 😀

Coding Challenge #2

Let's go back to Julia and Kate's study about dogs. This time, they want to convert dog ages to human ages and calculate the average age of the dogs in their study.

Your tasks:

Create a function `'calcAverageHumanAge'`, which accepts an arrays of dog's ages (`'ages'`), and does the following things in order:

1. Calculate the dog age in human years using the following formula: if the dog is `<= 2` years old, `humanAge = 2 * dogAge`. If the dog is `> 2` years old, `humanAge = 16 + dogAge * 4`
2. Exclude all dogs that are less than 18 human years old (which is the same as keeping dogs that are at least 18 years old)
3. Calculate the average human age of all adult dogs (you should already know from other challenges how we calculate averages 😉)
4. Run the function for both test datasets

Test data:

- Data 1: `[5, 2, 4, 1, 15, 8, 3]`
- Data 2: `[16, 6, 10, 5, 6, 1, 4]`

GOOD LUCK 😀

Coding Challenge #3

Rewrite the `'calcAverageHumanAge'` function from Challenge #2, but this time as an arrow function, and using chaining!

Test data:

- Data 1: `[5, 2, 4, 1, 15, 8, 3]`
- Data 2: `[16, 6, 10, 5, 6, 1, 4]`

GOOD LUCK 😀

Coding Challenge #4

Julia and Kate are still studying dogs, and this time they are studying if dogs are eating too much or too little.
Eating too much means the dog's current food portion is larger than the recommended portion, and eating too little is the opposite.
Eating an okay amount means the dog's current food portion is within a range 10% above and 10% below the recommended portion (see hint).

Your tasks:

1. Loop over the `'dogs'` array containing dog objects, and for each dog, calculate the recommended food portion and add it to the object as a new property. Do **not** create a new array, simply loop over the array. Forumla: `recommendedFood = weight ** 0.75 * 28`. (The result is in grams of food, and the weight needs to be in kg)
2. Find Sarah's dog and log to the console whether it's eating too much or too little. **Hint:** Some dogs have multiple owners, so you first need to find Sarah in the owners array, and so this one is a bit tricky (on purpose) 🤓
3. Create an array containing all owners of dogs who eat too much (`'ownersEatTooMuch'`) and an array with all owners of dogs who eat too little (`'ownersEatTooLittle'`).
4. Log a string to the console for each array created in 3., like this: *"Matilda and Alice and Bob's dogs eat too much!"* and *"Sarah and John and Michael's dogs eat too little!"*
5. Log to the console whether there is any dog eating **exactly** the amount of food that is recommended (just `true` or `false`)
6. Log to the console whether there is any dog eating an **okay** amount of food (just `true` or `false`)
7. Create an array containing the dogs that are eating an **okay** amount of food (try to reuse the condition used in 6.)
8. Create a shallow copy of the `'dogs'` array and sort it by recommended food portion in an ascending order (keep in mind that the portions are inside the array's objects 😉)

Hints:

- Use many different tools to solve these challenges, you can use the summary lecture to choose between them 😉
- Being within a range 10% above and below the recommended portion means: `current > (recommended * 0.90) && current < (recommended * 1.10)`. Basically, the current portion should be between 90% and 110% of the recommended portion.

Test data:

```
const dogs = [
  { weight: 22, curFood: 250, owners: ['Alice', 'Bob'] },
  { weight: 8, curFood: 200, owners: ['Matilda'] },
  { weight: 13, curFood: 275, owners: ['Sarah', 'John'] },
  { weight: 32, curFood: 340, owners: ['Michael'] },
];
```

GOOD LUCK 😀

Object Oriented Programming (OOP)

Coding Challenge #1

Your tasks:

1. Use a constructor function to implement a `'Car'`. A car has a `'make'` and a `'speed'` property. The `'speed'` property is the current speed of the car in km/h
2. Implement an `'accelerate'` method that will increase the car's speed by 10, and log the new speed to the console
3. Implement a `'brake'` method that will decrease the car's speed by 5, and log the new speed to the console
4. Create 2 `'Car'` objects and experiment with calling `'accelerate'` and `'brake'` multiple times on each of them

Test data:

- Data car 1: *'BMW'* going at 120 km/h
- Data car 2: *'Mercedes'* going at 95 km/h

GOOD LUCK 😀

Coding Challenge #2

Your tasks:

1. Re-create Challenge #1, but this time using an ES6 class (call it `'CarCl'`)
2. Add a getter called `'speedUS'` which returns the current speed in mi/h (divide by 1.6)
3. Add a setter called `'speedUS'` which sets the current speed in mi/h (but converts it to km/h before storing the value, by multiplying the input by 1.6)
4. Create a new car and experiment with the `'accelerate'` and `'brake'` methods, and with the getter and setter.

Test data:

- Data car 1: *'Ford'* going at 120 km/h

GOOD LUCK 😀

Coding Challenge #3

Your tasks:

1. Use a constructor function to implement an Electric Car (called `'EV'`) as a **child "class"** of `'Car'`. Besides a make and current speed, the `'EV'` also has the current battery charge in % (`'charge'` property)
2. Implement a `'chargeBattery'` method which takes an argument `'chargeTo'` and sets the battery charge to `'chargeTo'`
3. Implement an `'accelerate'` method that will increase the car's speed by 20, and decrease the charge by 1%. Then log a message like this: *'Tesla going at 140 km/h, with a charge of 22%'*
4. Create an electric car object and experiment with calling `'accelerate'`, `'brake'` and `'chargeBattery'` (charge to 90%). Notice what happens when you `'accelerate'`! **Hint:** Review the definiton of polymorphism 😉

Test data:

- Data car 1: *'Tesla'* going at 120 km/h, with a charge of 23%

GOOD LUCK 😀

Coding Challenge #4

Your tasks:

1. Re-create Challenge #3, but this time using ES6 classes: create an `'EVCl'` child class of the `'CarCl'` class
2. Make the `'charge'` property private
3. Implement the ability to chain the `'accelerate'` and `'chargeBattery'` methods of this class, and also update the `'brake'` method in the `'CarCl'` class. Then experiment with chaining!

Test data:

- Data car 1: *'Rivian'* going at 120 km/h, with a charge of 23%

GOOD LUCK 😀

Asynchronous JavaScript

Coding Challenge #1

In this challenge you will build a function `'whereAmI'` which renders a country **only** based on GPS coordinates. For that, you will use a second API to geocode coordinates. So in this challenge, you'll use an API on your own for the first time 😄

Your tasks:

PART 1

1. Create a function `'whereAmI'` which takes as inputs a latitude value (`'lat'`) and a longitude value (`'lng'`) (these are GPS coordinates, examples are in test data below).
2. Do "reverse geocoding" of the provided coordinates. Reverse geocoding means to convert coordinates to a meaningful location, like a city and country name. Use this API to do reverse geocoding: https://geocode.xyz/api. The AJAX call will be done to a URL with this format: https://geocode.xyz/52.508,13.381?geoit=json. Use the `fetch` API and promises to get the data. Do **not** use the `'getJSON'` function we created, that is cheating 😉
3. Once you have the data, take a look at it in the console to see all the attributes that you received about the provided location. Then, using this data, log a message like this to the console: *"You are in Berlin, Germany"*
4. Chain a `.catch` method to the end of the promise chain and log errors to the console
5. This API allows you to make only 3 requests per second. If you reload fast, you will get this error with code 403. This is an error with the request. Remember, `fetch()` does **not** reject the promise in this case. So create an error to reject the promise yourself, with a meaningful error message

PART 2

6. Now it's time to use the received data to render a country. So take the relevant attribute from the geocoding API result, and plug it into the countries API that we have been using.
7. Render the country and catch any errors, just like we have done in the last lecture (you can even copy this code, no need to type the same code)

Test data:

- Coordinates 1: 52.508, 13.381 (Latitude, Longitude)
- Coordinates 2: 19.037, 72.873
- Coordinates 3: -33.933, 18.474

GOOD LUCK 😀

Coding Challenge #2

For this challenge you will actually have to watch the video! Then, build the image loading functionality that I just showed you on the screen.

Your tasks:

Tasks are not super-descriptive this time, so that you can figure out some stuff by yourself. Pretend you're working on your own 😉

PART 1

1. Create a function `'createImage'` which receives `'imgPath'` as an input. This function returns a promise which creates a new image (use `document.createElement('img')`) and sets the `.src` attribute to the provided image path
2. When the image is done loading, append it to the DOM element with the `'images'` class, and resolve the promise. The fulfilled value should be the image element itself. In case there is an error loading the image (listen for the `'error'` event), reject the promise
3. If this part is too tricky for you, just watch the first part of the solution

PART 2

4. Consume the promise using `.then` and also add an error handler
5. After the image has loaded, pause execution for 2 seconds using the `'wait'` function we created earlier
6. After the 2 seconds have passed, hide the current image (set `display` CSS property to `'none'`), and load a second image (**Hint:** Use the image element returned by the `'createImage'` promise to hide the current image. You will need a global variable for that 😉)
7. After the second image has loaded, pause execution for 2 seconds again
8. After the 2 seconds have passed, hide the current image

Test data: Images in the `img` folder. Test the error handler by passing a wrong image path. Set the network speed to “Fast 3G” in the dev tools Network tab, otherwise images load too fast

GOOD LUCK 😀

Coding Challenge #3

Your tasks:

PART 1

1. Write an `async` function `'loadNPause'` that recreates Challenge #2, this time using `async/await` (only the part where the promise is consumed, reuse the `'createImage'` function from before)
2. Compare the two versions, think about the big differences, and see which one you like more
3. Don't forget to test the error handler, and to set the network speed to "Fast 3G" in the dev tools Network tab

PART 2

1. Create an async function `'loadAll'` that receives an array of image paths `'imgArr'`
2. Use `.map` to loop over the array, to load all the images with the `'createImage'` function (call the resulting array `'imgs'`)
3. Check out the `'imgs'` array in the console! Is it like you expected?
4. Use a promise combinator function to actually get the images from the array 😉
5. Add the `'parallel'` class to all the images (it has some CSS styles)

Test data Part 2: `['img/img-1.jpg', 'img/img-2.jpg', 'img/img-3.jpg']`. To test, turn off the `'loadNPause'` function

GOOD LUCK 😀