Homework #2

Higher Order Functions, Classes, Net Module

Overview

Repository Creation

👀 Click on this link: https://classroom.github.com/a/o0deJuFu (https://classroom.github.com/a/o0deJuFu) to accept this assignment in GitHub classroom.

- This will create your homework repository
- Clone it once you've created it.

Description

1. **hoffy.mjs** - write a series of functions that demonstrate the use of the rest operator (or call/apply), and higher order functions
2. **drawing.mjs** - create a class that represents svg markup
3. **sfmovie.mjs** and **report.mjs** - create functions and a short program to read data from a file and analyze that data using `map`, `reduce`, and `filter`

Submission Process

You will be given access to a private repository on GitHub. It will contain unit tests, stub files for your code, a `package.json` and a `.eslintrc`.

- The final version of your assignment should be in GitHub.
- **Push** your changes to the homework repository on GitHub.

(4 points) Make at Least 4 Commits

- Commit multiple times throughout your development process.
- Make at least 4 separate commits - (for example, one option may be to make one commit per part in the homework).

⚠️ ⚠️ ⚠️ Do not Use Loops! Use Return Values

For all parts, do not use:

- `while` loops
- `for` loops
- `for ... in` loops
- `for ... of` loops
- `forEach` method.

Of course, this means you'll be using some methods on Arrays

- `map`, `reduc` and `filter` will be helpful
- however, make sure to use their return value rather than just use each method as a mechanism for looping
  - for example: `const result = arr.map(fn)`
- you can use recursion as well

There will a small (-2) penalty every time one is used (with a cap of -10). (Homework is 100 points total)

Part 1 - Setup and Exercises

For this homework, you'll have files available in your repository, so you'll be cloning first.

Reminder: `for`, `while`, `for in`, `for of`, and `.forEach` are not allowed, and the return values of `map`, `reduce` and `filter` must be used.

The solutions to the following problems can go in the same file - **src/hoffy.mjs**:

Setup

1. using the url from your repository creation earlier…
2. clone with GitHub Desktop or the commandline client

```
git clone [YOUR REPO URL]
```

Background Info

Implement functions that use JavaScript features such as:

- the rest operator
- the spread operator
- functions as arguments
- functions as return values
- decorators
- optionally call/apply/bind
- optionally arrow functions
- Array methods: 'filter', 'map', 'reduce'

Go through the functions in order; the concepts explored build off one-another, and the functions become more and more challenging to implement as you go on.

Steps

1. prep…
    - ⚠️ **NOTE** that if the `.eslint.cjs` supplied is not working, please copy over the `.eslint.cjs` file from you previous assignment(s) (it's a hidden file and starts with `.`) … or see if there's an edstem thread on how handle the issue
    - create a `.gitignore` to ignore node_modules
    - create a `package.json` by using `npm init`
    - make sure that `mocha`, `chai`, and `eslint` are still installed (similar to previous assignment)

    ```
    npm install --save-dev mocha
    npm install --save-dev eslint
    npm install --save-dev chai
    npm install --save-dev eslint-plugin-mocha
    ```

    - you'll also need a few additional modules installed locally for the unit tests to run:
        - finally, install sinon and mocha-sinon locally for mocking `console.log` (these are for unit tests)
        - `npm install --save-dev sinon`
        - `npm install --save-dev mocha-sinon`
2. implement functions below in **hoffy.mjs**
3. make sure you export your functions as you implement them so that…
4. you can run tests as you develop these functions (again, the tests are included in the repository): `npx mocha tests/hoffy-test.mjs`
5. also remember to run eslint (there's a `.eslintrc` file included in the repository): `npx eslint src/*`

(40 points) Functions to Implement

(-2 per while, for, forEach, for of, or for in loop used)

`getEvenParam(s1, s2, s3 to sN)`

Parameters:

- `s1, s2, s3 to sN` - any number of string arguments

Returns:

- an array of parameters that have even index (starting from 0)
- if no arguments are passed in, give back an empty array

Description:

Goes through every string argument passed in and picks up the even index parameters (suppose the first parameter has index 0, second has index 1, and so on). No error checking is required; you can assume that every argument is a string or no arguments are passed in at all. If there are no arguments passed in, return an empty array

HINTS:

- use `rest` parameters!

Example:

```
getEvenParam('foo', 'bar', 'bazzy', 'qux', 'quxx') // --> ['foo', 'bazzy', 'quxx']
getEvenParam('foo', 'bar', 'baz', 'qux') // --> ['foo', 'baz']
getEvenParam() // --> []
```

`myFlatten(arr2d)`

Parameters:

- `arr2d` - a 2-dimensional array (an array composed of sub-arrays, such as `[['a', 'b'], ['c'], ['d', 'e', 'f']]`); assume just one level of nesting

Returns:

- a new 1-dimensional array

Description:

Without using the built in `flat` or `flatMap` methods, implement a function that "flattens" a 2-dimensional array so that all of the elements of the subarray become separate elements in a new 1-dimensional array. Only handle 1 level of nesting (no need to handle higher dimensional arrays).

Example:

```
const arr = myFlatten([[1, 2, 3], [4, 5]])
// arr is [1, 2, 3, 4, 5]
```

`maybe(fn)`

Parameters:

- `fn` - the function to be called

Returns:

- a new `function` or `undefined` - the `function` calls the original function

Description:

`maybe` will take a function, `fn` and return an entirely new function that behaves mostly like the original function, `fn` passed in, but will return undefined if any `null` or `undefined` arguments are passed in to `fn`.

The new function will take the same arguments as the original function (`fn`). Consequently when the new function is called, it will use the arguments passed to it and call the old function and return the value that's returned from the old function. However, if any of the arguments are `undefined` or `null`, the old function is not called, and `undefined` is returned instead. You can think of it as a way of calling the old function only if all of the arguments are not `null` or not `undefined`.

Example:

```
function createFullName(firstName, lastName) {
    return `${firstName} ${lastName}`;
}
maybe(createFullName)('Frederick', 'Functionstein'); // Frederick Functionstein
maybe(createFullName)(null, 'Functionstein');        // undefined
maybe(createFullName)('Freddy', undefined);          // undefined
```

`filterWith(fn)`

Parameters:

- `fn` - a *callback* function that takes in a single argument and returns a value (it will eventually operate on every element in an array)

Returns:

- `function` - a function that…
  - has 1 parameter, an `Array`
  - returns a new Array where only elements that cause `fn` to return `true` are present (all other elements from the old Array are not included)

Description:

This is different from regular filter. The regular version of filter immediately calls the callback function on every element in an Array to return a new Array of filtered elements. `filterWith`, on the other hand, gives back a function rather than executing the callback immediately (think of the difference between bind and call/apply).

`filterWith` is basically a function that turns another function into a filtering function (a function that works on Arrays).

Example:

```
// original even function that works on Numbers
function even(n) {return n % 2 === 0;}

// create a 'filter' version of the square function
filterWithEven = filterWith(even);

// now square can work on Arrays of Numbers!
console.log(filterWithEven([1, 2, 3, 4])); // [2, 4]

const nums = [1, NaN, 3, NaN, NaN, 6, 7];
const filterNaN = filterWith(n => !isNaN(n));
console.log(filterNaN(nums)); // [1, 3, 6, 7]
```

`repeatCall(fn, n, arg)`

Parameters:

- `fn` - the function to be called repeatedly
- `n` - the number of times to call function, `fn`
- `arg` - the argument to be passed to `fn`, when it is called

Returns:

- `undefined` (no return value)

Description:

This function demonstrates using functions as an argument or arguments to another function. It calls function, `fn`, `n` times, passing in the argument, `arg` to each invocation / call. It will ignore the return value of function calls. Note that it passes in only one `arg`.

Hint:

Write a recursive function within your function definition to implement repetition.

Examples:

```
repeatCall(console.log, 2, "Hello!");
// prints out:
// Hello!
// Hello!

// calls console.log twice, each time passing in only the first argument

repeatCall(console.log, 2, "foo", "bar", "baz", "qux", "quxx", "corge");

// prints out (again, only 1st arg is passed in):
// foo
// foo
```

`limitCallsDecorator(fn, n)`

Parameters:

- `fn` - the function to modify (*decorate*)
- `n` - the number of times that `fn` is allowed to be called

Returns:

- `function` - a function that…
  - does the same thing as the original function, `fn` (that is, it calls the original function)
  - but can only be called `n` times
  - after the `n` th call, the original function, `fn` will not be called, and the return value will always be `undefined`

Description:

You'll read from a variable that's available through the closure and use it to keep track of the number of times that a function is called… and prevent the function from being called again if it goes over the `max` number of allowed function calls. Here are the steps you'll go through to implement this:

1. create your decorator (function)
2. create a local variable to keep track of the number of calls
3. create an inner function to return

    - the inner function will check if the number of calls is less than the max number of allowed calls
    - if it's still under max, call the function, `fn` (allow all arguments to be passed), return the return value of calling `fn`, and increment the number of calls
    - if it's over max, just return `undefined` immediately without calling the original function

Example:

```
const limitedParseInt = limitCallsDecorator(parseInt, 3);
limitedParseInt("423") // --> 423
limitedParseInt("423") // --> 423
limitedParseInt("423") // --> 423
limitedParseInt("423") // --> undefined
```

`myReadFile(fileName, successFn, errorFn)`

Parameters:

- `fileName` - a `string`, the name of the file to read from
- `successFunc` - a `function` to call if file is successfully read
    - function has single argument, the data read from the file as a `string`
- `errorFunc` - a `function` to call if error occurs while reading file (such as file does not exist)
    - function has single argument, an error object (the same error object passed to `fs.readFile`'s callback)

Returns:

- `undefined` (no return value)

Description:

This function gives us an alternative *interface* to `fs.readFile`. `fs.readFile` typically takes a single callback (after the file name) with two arguments: an error object and the data read from the file. This function, instead, takes two callbacks as arguments (both after the file name) – one to be called on success and one to be called on failure. Both callbacks only have one parameter (the data read from the file or an error object). The actual implementation simply calls `fs.readFile`. Note that you can assume that file read in is text, so pass in a second argument to `fs.readFile` to read the data as utf-8: `fs.readFile('filename.txt', 'utf-8', callback)`

⚠️ **WARNING** - the test file was created using `\n` as the line break. If you're on windows, you can modify the test to use `\n` instead of `os.EOL` so that the tests are hardcoded to use `\n` rather than a platform dependent line break (`\r\n` on windows). This makes the tests less portable, but it will pass based on the included text file.

Example:

```
// Assuming the contents of the file, tests/words.txt is:
// ant bat
// cat dog emu
// fox
const success = (data) => console.log(data);
const failure = (err) => console.log('Error opening file:', err);

myReadFile('tests/words.txt', success, failure);

// prints out:
// ant bat
// cat dog emu
// fox

myReadFile('tests/fileDoesNotExist.txt', success, failure);
// The error message would be printed out because the file does not exist:
// Error opening file: ...
```

`rowsToObjects(data)`

Parameters:

- `data` - an `object` with the following properties:
    - `headers` - the names of the columns of the data contained in `rows`
    - `rows` - a 2-dimensional `Array` of data, with the first dimension being rows, and the second being columns

Returns:

- an `Array` of objects with the original headers (column names) as properties… and values taken from the original data in each row that aligns with the column name

Description:

This converts a 2-d array of data:

```
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

…into an Array of objects with property names, given the property names as headers.

```
// headers
['a', 'b', 'c'];
```

```
// result
// [{a: 1, b: 2, c: 3}, {a: 4, b: 5, c: 6}, {a: 7, b: 8, c: 9}];
```

The data should come in as an object where the headers an `Array` in the `headers` property of the object, and the data is the value in the `rows` property of the object:

```
{
        headers: ['col1', 'col2'],
        rows: [[1, 2],
               [3, 4]]
}
```

Example:

```
const headers = ['a', 'b', 'c'];
const rows = [[1, 2, 3], [4, 5, 6], [7, 8, 9]];
const result = rowsToObjects({headers, rows})
console.log(result);
// [{a: 1, b: 2, c: 3}, {a: 4, b: 5, c: 6}, {a: 7, b: 8, c: 9}];
```

Test, Lint and Commit

Once you're finished with your functions, remember to:

1. make sure all tests are passing
2. make sure that eslint shows no errors
3. commit and push your code!

Part 2 - Drawings and Classes

For this part of the homework, create classes in the file in your repository called **src/drawing.mjs**.

SVG Overview

You'll be implementing classes that represent drawings / parts of a drawing by using SVG.

SVG (https://developer.mozilla.org/en-US/docs/Web/SVG) is a markup language for creating vector graphics.

You can think of it as HTML for drawing! (Pssst… it's actually XML). So, some terms used to describe HTML and XML will describe SVG as well:

- **element** - is a part of an SVG image; it's composed to tags, attributes and possibly text / content
- **tag** - is the name of an element surrounded by < and > `<rect>`; there can be opening and closing tags in an element: `<rect></rect>`
- **attribute** - name and value pairs within an opening tag `<rect width="100">`
    - note that for the example above, the name is unquoted
    - and the value is quoted with double quotes
- **content** - the text surrounded by tags: this is content

Elements can be nested within elements. For example: `<svg xmlns="http://www.w3.org/2000/svg"> <rect width="50" height="50"></rect> </svg>`

SVG markup can be placed in a `.svg` file. An `.svg` file can be opened with browsers, such as Chrome or Firefox. Some file explorers even render SVG directly. SVG can even be embedded directly in an html document (view source on this page, and search for `svg` tags.

`svg` Element

The container (root element) for an SVG drawing is (big surprise) `svg`:

```
<svg xmlns="http://www.w3.org/2000/svg">
</svg>
```

- within this container, the x values increase from left to right.
- the y values increase from top to bottom

`rect` Element

To draw a rectangle, use the `rect` element. Some attributes that you can define on a `rect` element are:

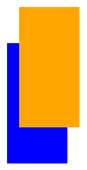
- `x` - the x location of the upper left corner of the rectangle
- `y` - the y location of the upper left corner of the rectangle
- `width` - the width in pixels of the rectangle
- `height` - the height in pixels of the rectangle
- `fill` - the fill color of the rectangle; this can be text, such as `"red"`, `"orange"`, etc.

Note that the `rect` element can be self-closing, but for simplicity in our implementation, we'll create `rect` elements with both an opening and closing tag.

Here's an example of drawing two rectangles:

```
<rect x="25" y="50" width="50" height="100" fill="blue"></rect>
<rect x="35" y="20" width="50" height="100" fill="orange"></rect>
```

The code above results in this drawing:



`text` Element

The `text` element allows you to place text in your drawing. Some attributes include:

- `x` - the x location of the upper left corner of the text
- `y` - the y location of the upper left corner of the text
- `fill` - the fill color of the text; this can be a color name, such as `"red"`, `"orange"`, etc.
- `font-size` - the size of the text

Note that the content of a text element (the value between the `text` tags) is what will be displayed. Here's an example:

```
<text x="50" y="70" fill="blue" font-size="70">SVG FTW!</text>
```

The code above results in the following drawing:

Implementation

You'll be creating classes that represent generic svg elements, `svg`, `rect`, and `text`. Your implementation:

- **does not require type checking of incoming arguments**
- **should not use**

- `while` loops
- `for` loops
- `for ... in` loops
- `for ... of` loops
- `forEach` method

There will a small (-2) penalty every time one is used. (Homework is 100 points total)

Create these four classes in `src/drawing.mjs` **and export them**:

1. `GenericElement(name)` - represents a generic SVG element
   - `name` - the name of the element (this name is freeform, so, for example: `svg`, `rect`, or even an element that we haven't covered… `circle`)
2. `RootElement()` - represents an `svg` element
   - should have the attribute: `xmlns="http://www.w3.org/2000/svg"` …so that it is treated as a valid SVG (XML)
3. `RectangleElement(x, y, width, height, fill)` - represents a `rect` element

    - `x` - the value for the `x` attribute of this `rect` element
    - `y` - the value for the `y` attribute of this `rect` element
    - `width` - the value for the `width` attribute of this `rect` element
    - `height` - the value for the `height` attribute of this `rect` element
    - `fill` - the value for the `fill` attribute of this `rect` element as a color string, such as `"red"`, `"green"`, etc.
4. `TextElement(x, y, fontSize, fill, content)` - represents a `text` element
    - `x` - the value for the `x` attribute of this `text` element
    - `y` - the value for the `y` attribute of this `text` element
    - `fontSize` - the value for the `font-size` attribute of this `text` element
    - `fill` - the value for the `fill` attribute of this `text` element as a color string, such as `"red"`, `"green"`, etc.
    - `content` - content between the `text` tags; the *actual* text

Somewhere in these classes, you should define the following methods (you're free to implement these methods in whatever classes make sense, as long as the example code below works):

- `addAttr(name, value)` - adds an attribute named, `name` to the element called on, with value, `value`
    - `name` - the name of the attribute to be added
    - `value` - the value of the attribute to be added
- `setAttr(name, value)` - set an attribute, `name` to the element called on, with value, `value`. Assume the attribute name always exists when this function is called.
    - `name` - the name of the attribute to be set
    - `value` - the value of the attribute to be set
- `addAttrs(obj)` - adds attributes (keeps existing attributes if already present, but if attribute names are the same, your discretion) to the element called on by using the property names and values of the object passed in: `{x: 100, y: 5}` would translate to `x="100" y="5"`
    - `obj` - the object containing properties and value that will be used to create attributes and values on the element
- `removeAttrs(arr)` - remove attributes (remove existing attributes if already present, but if attribute names don't exist, do nothing) to the element called on by using the property names array passed in: `['x','y','z','a']` would remove all attributes `x,y,z,a`, if the svg has these attributes.
    - `arr` - an array containing properties that will be removed
- `addChild(child)` - adds an element as a child to the element that this is called on (that is, `child`, is nested in this element
    - `child` - the element to be nested in the parent element
- `toString()` - returns the string representation of this element, including tags, attributes, text content, and nested elements…
    - for example `'<svg xmlns="http://www.w3.org/2000/svg"><text x="50" y="70" fill="blue" font-size="40">SVG FTW!</text> </svg>'`
    - you can use any whitespace at your discretion (for example, you can add newlines if you want)
- `write(fileName, cb)` - write the string representation of your element and all its children to a file called `fileName`
    - `fileName` - the path of the file to write to
    - `cb` - function to call when writing is done
        - the callback function can have no parameters
    - use `fs.writeFile` (https://nodejs.org/api/fs.html#fs_fs_writefile_file_data_options_callback) as part of your implementation

Your implementation must follow the requirements listed below:

- you must use `extends` for one or more of your classes
- you can add any properties necessary to make your code work
- an element's attributes can be in any order when the markup is generated (no specific ordering is required from your implementation)
- an element's children can be in any order when the markup is generated (no specific ordering is required from your implementation)
- you can place methods in whatever class you think is appropriate, and you can even repeat code (though you can use inheritance to prevent this)
- finally, once you're done, copy the code below into the same file
- run your file…

```
// create root element with fixed width and height
const root = new RootElement();
root.addAttrs({width: 800, height: 170, abc: 200, def: 400});
root.removeAttrs(['abc','def', 'non-existent-attribute']);

// create circle, manually adding attributes, then add to root element
const c = new GenericElement('circle');
c.addAttr('r', 75);
c.addAttr('fill', 'yellow');
c.addAttrs({'cx': 200, 'cy': 80});
root.addChild(c);

// create rectangle, add to root svg element
const r = new RectangleElement(0, 0, 200, 100, 'blue');
root.addChild(r);

// create text, add to root svg element
const t = new TextElement(50, 70, 70, 'red', 'wat is a prototype? 😬');
root.addChild(t);

// show string version, starting at root element
console.log(root.toString());

// write string version to file, starting at root element
root.write('test.svg', () => console.log('done writing!'));
```

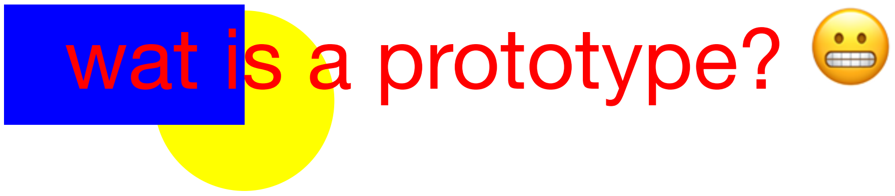
The code above above should produce the markup below by writing to a file, `test.svg`:

```
<svg xmlns="http://www.w3.org/2000/svg" width="800" height="170">
<circle r="75" fill="yellow" cx="200" cy="80">
</circle>
<rect x="0" y="0" width="200" height="100" fill="blue">
</rect>
<text x="50" y="70" fill="red" font-size="70">wat is a prototype? 😬
</text>
</svg>
```

Remember that in the resulting markup:

- spacing is at your discretion: you can add / remove newlines, for example
- an element's attributes can be in any order (your discretion)
- an element's children can be in any order (your discretion)

When opened with your web browser (like Chrome or Firefox), you should see:



Troubleshooting

- if your browser does not display the image above…
- make sure that the markup produced by your code matches
- (check each attribute, value, etc.)

Part 3 - Processing San Francisco Movie Data

As with previous parts, `for`, `while`, `for in`, `for of`, and `.forEach` are not allowed, and the if using `map`, `reduce` and `filter`, the return values must be used.

In this part, you'll work with a dataset of locations and movies filmed at those locations, all within San Francisco. You will read the data from a csv file and convert it into objects in JavaScript. You'll print out a report that analyzes the data by using functions that you've implemented.

You'll be using two files for this:

1. `sfmovie.mjs` to create a library of functions
2. `report.mjs` to read the file, use your module of helper functions, and print out a report

You'll have to work with both files simultaneously. `sfmovie.mjs` will contain helper functions, while `report.mjs` will read the data and – using the functions you've created – print our a report.

Importing Data

Download the data by going to https://data.sfgov.org/Culture-and-Recreation/Film-Locations-in-San-Francisco/yitu-d5am (https://data.sfgov.org/Culture-and-Recreation/Film-Locations-in-San-Francisco/yitu-d5am) and clicking the "Export" button on the top right, then the "CSV" button.

- place it in `data` folder (create this folder if it isn't present)
- in `report.mjs` , write code to read and parse the file
- start by reading in the file (which should have a filename something like `Film_Locations_in_San_Francisco.csv`)
- make sure that `fs` the module is brought in using ES Modules / `import` , then call the function)
- **do not use readFileSync**
- see the docs on `fs.readFile` (hint: make sure you specify `utf8` as the second argument)
- the file that you read in contains a list of movies filmed in San Francisco
    - there's one movie per line
    - each movie is represented by a comma-delimited string

Parse the data by installing the csv-parse (https://csv.js.org/parse/) library: `npm install csv-parse`

- this library, `csv-parse` , supports callback based parsing (as well as streaming, promise and sync)
    - again, you can install in the root of your project with `npm install csv-parse`
    - see the docs for `csv-parse` here (https://csv.js.org/parse/api/callback/)
- add your parsing code within the callback to `readFile` so that it only runs when the file is successfully read
- use the callback based version of `parse` (https://csv.js.org/parse/api/callback/)
- for example:

import { parse } from 'csv-parse' // within the callback to readFile, where data is the data read from the file parse(data, {}, (err, parsedData) => { // parseData is the data read form the file })

Convert each line into an object:

- the properties correspond with the first row of the file (the headers) HINT: use `rowsToObjects(data)` function
- each resulting object is placed into an `Array` for later processing
- you can check out some String (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String) and Array (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array)
- note that the goal of this assignment is to work with higher order functions, so memory efficiency does not need to be taken into consideration
- all of your processing can only be done from within the callback function for `parse` (which itself is in the callback to `readFile`)
- examine the resulting `Array` … (for example, try printing it out!)

Examining the Data

- assuming that your parsed data is in a variable called `data` …
- `data` will be an `Array` of objects (each line should have been convert into an object before adding into the `data` array)
- each object in `data` contains the column attributes and corresponding values
- if there are any parsing issues, you can skip those specific lines

Below is a sample listing from the dataset with a description of properties you'd need for your assignment

```
{
    Title: "A Jitney Elopement",
    "Release Year": "1915",
    Locations: "20th and Folsom Streets",
    "Fun Facts": "",
    "Production Company": "The Essanay Film Manufacturing Company",
    Distributor: "General Film Company",
    Director: "Charles Chaplin",
    Writer: "Charles Chaplin",
    "Actor 1": "Charles Chaplin",
    "Actor 2": "Edna Purviance",
    "Actor 3": ""
}
```

Note that movies will appear multiple times, as they can be filmed in multiple locations.

Helper Functions in `sfmovies.mjs`

Now, switch over to your `sfmovies.mjs` module; you'll define your functions here. Each function will have at least one parameter, `data`, that's in the format of an `Array` of objects, with each object representing a filming location and movie (see sample object above).

1. `longestFunFact(data)` - gives back the whole `object` that has the longest fun fact. (you can just count length of the string in `Fun Facts` column) If there's a tie, return any of them.
    - `longestFunFact(data)`
    - example return value:
    ```
    {
      Title: 'Etruscan Smile',
      'Release Year': '2017',
      Locations: 'Legion of Honor, 100 34th Ave',
      'Fun Facts': 'California Palace of the Legion of Honor was completed in 1924, and on Armistice Da
    y of that year the doors opened to the public. In keeping with the wishes of the donors, to “honor
    the dead while serving the living,” it was accepted by the city of San Francisco as a museum of fin
    e arts dedicated to the memory of the 3,600 California men who had lost their lives on the battlefi
    elds of France during World War I.',
      // additional properties after
    }
    ```
2. `getTitlesByYear(data, year)` - extracts the **unique** titles of all movies released in the year(as a Number) given
    - returns titles of all movies followed by year in parentheses
    - the entire output should be in uppercase
    - no duplicate titles
    - to remove duplicates, you can use `filter` so that the callback takes both the element and the `index` … then use `indexOf` which finds the first occurrence
    - `.filter((s, i, arr) => arr.indexOf(s) === i)`
    - alternatively, use a JavaScript set (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Set)
    - example return value:
    ```
    [
      'I'M A VIRGO (2023).',
      'BLINDSPOTTING (SEASON 2) (2023).',
      'THE LAST THING HE TOLD ME (2023).',
      'ANT-MAN AND THE WASP: QUANTUMANIA (2023).',
      'EARTH MAMA (2023).'
    ]
    ```
3. `actorCounts(data)` - gives back an object that contains actor names as keys and the number of times they appear in the dataset as the value
    - an actor can appear multiple times for the same movie, as a movie can appear in the dataset associated with different locations
    - returns an object containing counts
    - `actorCounts(data)`
    - example return value
    ```
    {
      'Hugh Laurie': 114,
      'Ethan Suplee': 114,
      'Gretchen Mol': 84,
      'Frankie Alvarez': 84
    }
    ```

Analytics / Back to `report.mjs`

Once you've implemented helper functions in `sfmovie.mjs`, you can use those functions in `report.mjs`. If it's helpful, you're free to reuse functions from `hoffy.mjs`.

Again, go back `report.js` …

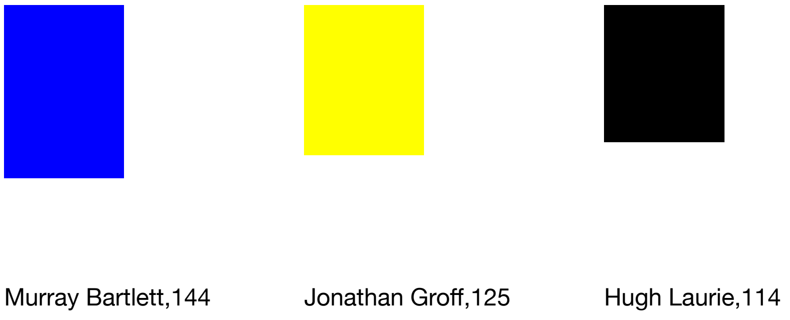
1. you should already have code present for reading and parsing the file
2. refactor this code so that the path is not hardcoded…
3. use `process.argv[2]` to retrieve the path specified when `report.mjs` is run on the commandline:
    - `node src/report.mjs /path/to/file`
    - `process.argv[2]` will contain `/path/to/file`
    - running your report from the root of your project will likely look like this:
    - `node src/report.mjs data/Film_Locations_in_San_Francisco.csv`
4. bring in the functions from `sfmovie.mjs` using imports
    - again feel free to add more helper functions / utilities, like your functions in `hoffy.mjs`)
5. print out the entire movie object with the longest Fun Fact

6. print out a list of movies released in 2023
7. draw an SVG that shows the top 3 popular actors and their occurrences (like a histogram)
    - because your helper founct, `actorCount` returns an object of counts, you might find the following methods functions / useful:
        - Object.entries (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/entries)
        - sort (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/sort) (using the compare function variant)
    - the resulting svg should contain 3 rectangle objects
    - the text should contain actor names, and number of occurrences
    - the width should be the same, but the height should be the occurrences of each actor
    - output the SVG in a file named `actors.svg`
    - it's ok if the graph is upside down (because we draw the rectangles from top to bottom)
    - **when you import drawing.mjs to draw svg, it's okay if your test code is still present and regenerates your testing svg**

The resulting output should look like this:

```
{
  Title: 'Etruscan Smile',
  'Release Year': '2017',
  Locations: 'Legion of Honor, 100 34th Ave',
  'Fun Facts': 'California Palace of the Legion of Honor was completed in 1924, and on Armistice Day of that
year the doors opened to the public. In keeping with the wishes of the donors, to “honor the dead while servi
ng the living,” it was accepted by the city of San Francisco as a museum of fine arts dedicated to the memory
of the 3,600 California men who had lost their lives on the battlefields of France during World War I.',
  'Production Company': 'Po Valley Productions, LLC',
  Distributor: 'TBD',
  Director: 'Oded Binnun/ Michel Brezis',
  Writer: 'Sarah Bellwood, Michal Lali Kagan, Michael McGowan Amital Stern, Jose Luis Sampedro',
  'Actor 1': 'Brian Cox',
  'Actor 2': 'Roseanne Arquette',
  'Actor 3': 'Thora Birch',
  'SF Find Neighborhoods': '7',
  'Analysis Neighborhoods': '17',
  'Current Supervisor Districts': '4'
}
[
  'I’M A VIRGO (2023).',
  'BLINDSPOTTING (SEASON 2) (2023).',
  'THE LAST THING HE TOLD ME (2023).',
  'ANT–MAN AND THE WASP: QUANTUMANIA (2023).',
  'EARTH MAMA (2023).'
]
[
  [ 'Murray Bartlett', 144 ],
  [ 'Jonathan Groff', 125 ],
  [ 'Hugh Laurie', 114 ],
]
```

as well as a svg file named `actors.svg`, showing the top 3 actors and their occurrences like below:



Lint, commit and push your code; the next part will make modifications to this existing code (you can overwrite your work in this file directly for the next part).