CS312 - Programming Assignment 1

Initial Due Date: 2023-02-23 8:15AM
Final Due Date: 2023-03-31 4:15PM

Goals

- Develop a basic familiarity with `git` and GitHub classroom
- Get started with basic JavaScript and Node.js
- Practice some of the functional aspects of JavaScript (higher-order functions and closures)
- Practice test-driven development (TDD)
- Use a linter to write more consistent, more maintainable, higher quality, code

Prerequisites

1. Install `git` and Node.js as described on the Getting Started page
2. Click the GitHub classroom link and then clone repository GitHub classroom creates to your local computer (in your shell, execute 💻 `git clone` followed by the address of the repository).
3. Update the `package.json` file with your name and e-mail
4. Install the package dependencies with 💻 `npm install`

Background

Running and Testing Your Program

As with the practical, all of the function definitions in the starter code are being exported from the module.

Start the Node.js REPL by executing 💻 `node` in your shell. You can then load the contents of the module with:

```
let pa1 = await import("./index.js");
```

Your functions will now be properties of the `pa1` object. For example, to run the `myMax` function, you can type `pa1.myMax([4,2,6])`.

You can and are encouraged to practice test-driven development, TDD, (as seen in class). The assignment skeleton is set up for unit testing with Jest. You will find a variety of tests, including the assignment examples in `index.test.js`.

To run the tests, run 💻 `npm test`.

Note that these tests are currently failing because you haven't written any code yet. I suggest that you go through `index.text.js` and change all of the `describe()` calls to `describe.skip()`. This tells jest to skip the tests in that test suite (you can also use the same technique to skip individual tests). As you work on each section, remove the `skip` to get feedback on the function as you go. You are welcome to add more tests to help you debug particular behaviors.

Code that fails any test does not yet meet the specifications and is incomplete.

Motivating Git

Git is a distributed version control system (VCS). Git, and its "killer app" GitHub, will play a key role in our workflow. At its simplest, a VCS enables us to "checkpoint" our work (a *commit* in Git terminology), creating a distinct version of our codebase that that we can return to. The opportunity to track and undo changes makes it easier to find new bugs (by reviewing differences to a prior working version), maintain a clean code base (we don't need to keep commented out code around in case we need it again), confidently make large changes knowing we can always recover a working version, and much more. For these reasons and more solo developers will use a VCS (and so should you!), but it is even more useful in a team environment.

How will you know if you and another developer modify the same file (in potentially incompatible ways)? How do you ensure you don't end up with a teammate's half-working modifications? We will use the VCS to prevent these problems.

The "distributed" in "distributed VCS" means that no central repository is required. Each Git repository contains the entire history of the project and thus each developer can work independently, making commits (checkpoints) without interfering with anyone else. Only when you *push* or *pull* those changes to another copy of the repository do your changes become visible to anyone else. Further we will use branches to segregate our changes. A branch is just a parallel version of the codebase. This allows you to experiment, while leaving the *master* branch untouched until your new feature is ready to be *merged* back into the master.

Git does not require a central repository. However, teams still tend to use a central repository to facilitate their work (we will use GitHub in this role). There isn't anything technically special about the shared repository other than that the team decides to share their changes via that central repository rather than directly with each other.

Hopefully this short (and incomplete) description provides some context on why we use VCS. Please refer to various online resources for more details on how to use Git (and GitHub) most effectively.

Assignment

Part 1: Reduce

Write a function `myMax(arr)` to find the largest value in an array using `reduce`. It should accept an array as an argument and return the largest value in the array (you can assume that the array is non-empty and that the values in the array are comparable). For example, `myMax([1, 2, 3])` should return `3`. Your code should be of the form `const myMax = arr => arr.reduce(TODO);`, where `TODO` should be replaced with the actual functionality. Your solution should *not* make use of `Math.max`.

Part 2: Filtering

Write a function `threshold(objs, field, cutoff)`. This function takes in an array of objects (`objs`), the name of a property found in the objects (`field`), and a cutoff value (`cutoff`). The function should return an array of those objects in `objs` whose values for `field` are less than or equal to `cutoff`. For example, `threshold([{x: 4, y: 5}, {x: 2, y: 9}, {x: 1, y: 1}], 'y', 5)` should return `[{x: 4, y: 5}, {x: 1, y: 1}]`. Your solution must use the array's filter method.

Part 3: Mapping

Write a function `parseEmails(strings)`. This function takes in an Array of strings (`strings`), where each string is expected to be in the format `'First Last <Email>'` (to keep this simple, we will assume names with a single given name followed by a family name). So, for example, my name and address would look like `'Michael Linderman <mlinderman@middlebury.edu>'`. For each such string, the function should return a JavaScript object with fields for `first`, `last`, and `email`. My string should be transformed to the object `{first:'Michael', last: 'Linderman', email:'mlinderman@middlebury.edu'}` (note that the '<' and '>' have been stripped from the email address).

Some additional requirements:

1. Your solution must use the `map` function.
2. While the function is designed to accept an array of strings, it should *also* accept a single string. Your function will need to detect this and do the right thing (the output will always be an Array, however). *I suggest thinking about how to handle the single string case, and then applying that solution to the array case when it occurs...*
3. If a string is malformed, instead of returning an object, the function should return `null` (we are only worried about the structure, you don't need to validate the email address in any way). So, for example, `parseEmails(["Jodi Whittaker <jwhittaker@prydon.edu>", "Peter Capaldi pcapaldi@prydon.edu"])` should return `[{first:'Jodi', last:'Whittaker', email:'jwhittaker@prydon.edu'}, null]`.

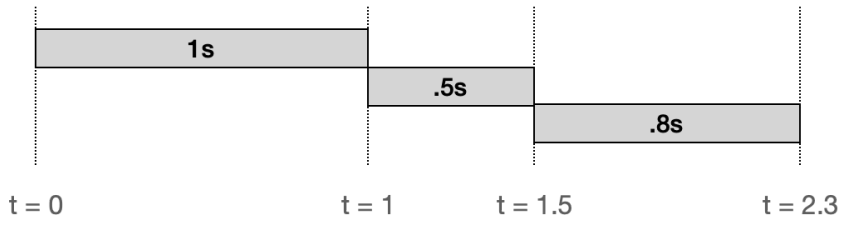
Part 4: Interval alarm with closures

You are building an application to facilitate interval workouts. Write a function `intervalAlarm` that takes an array of integers specifying interval times in seconds and returns a function that you can invoke to start the timer. The returned function should not take any arguments. When you invoke this function it should print a message when each interval expires, like shown below (including the length of the specified interval and the total time elapsed). Invoking `intervalAlarm` should not start the timer. You will need to use a closure.

```
> const alarm = intervalAlarm([1, 0.5, .8])
undefined
> alarm()
undefined
> Interval of 1s completed (1.006s elapsed)!
Interval of 0.5s completed (1.502s elapsed)!
Interval of 0.8s completed (2.304s elapsed)!

> alarm()
undefined
> Interval of 1s completed (1s elapsed)!
Interval of 0.5s completed (1.501s elapsed)!
Interval of 0.8s completed (2.302s elapsed)!
```

`intervalAlarm([1, 0.5, .8])`



Note that when you run the function via Jest the timing will not be correct because the tests mock out the timer.

Part 5: Calendar histogram

Now that you have experience with data structures and iteration, we will combine those tools to implement a calendar "histogram". You are trying to find the number of individuals who are available in specific windows during the week (think Doodle). A window is specified by an integer day of the week (0 is Sunday, 6 is Saturday), an inclusive start time and an exclusive end time (time is expressed in minutes since midnight). For example Tuesday 8:00AM-9:15AM would be specified as the following object:

```
{
  day: 2,
  start: 480,
  end: 555
}
```

Write a function `availablityCount(windows, availabilities)` that has two arrays of time window objects as arguments. Your function should return a *deep copy* of the `windows` array (in any order) with each window also containing a `count` field of the number of objects in `availabilities` that overlap that window. This needs to be a *deep copy*, that creates copies of the window objects as well to avoid changing the original data. Windows in which no one was available should have a count of zero. You should only increment the count if the availability fully overlaps the time window. For example the following call

```
availabilityCounts(
  [{ day: 2, start: 480, end: 495 }, { day: 2, start: 840, end: 855 }],
  [{ day: 2, start: 480, end: 555 }]
)
```

should return

```
[{ day: 2, start: 480, end: 495, count: 1 }, { day: 2, start: 840, end: 855, count: 0 }]
```

because the availability (Tuesday 8:00AM-9:15AM) fully overlaps the first window, but does not overlap the second (Tuesday 2:00PM - 2:45PM).

There are many possible approaches to this problem. Any correct implementation will meet expectations (e.g. $O(n^2)$) time complexity is acceptable), but for context, the solution is less than 20 nicely formatted, heavily commented, lines (think about how you could use the functional tools you worked with above). An exemplary solution will be easily understandable and maintainable (i.e., reflect "strategic" vs. "tactical" software development).

Finishing Up

Once your program is working make sure you don't have any style issues by running ESLint via 💻 `npm run lint`. ESLint can fix many of the errors automatically by running 💻 `npm run lint -- --fix` (although since ESLint can sometimes introduce errors during this process, we suggest committing your code before running "fix" so you can rollback any changes). To get full credit your code must have zero ESLint errors or warnings.

It is OK to deactivate a rule for a specific line as long as doing so is a considered decision (not a means to hammer ESLint into submission). You will need to justify each deactivation with a comment. For example you will need to deactivate the console warning for your `intervalAlarm` function.

```
// eslint-disable-next-line no-console
console.log(...)
```

If you look closely at the `package.json` file, you will notice additional packages and scripts for running the Prettier code formatting tool automatically when you commit a file. We use Prettier to ensure a consistent code formatting, automatically, without any nitpicking. Prettier will run automatically, or you can run it manually with `npx prettier --write *.js`. We will use Prettier extensively throughout the semester.

Notice that there is an additional file in your directory, `.gitignore`, which specifies files that should not be tracked by Git. It is good practice to create this file first in your repository to prevent undesired files from getting committed. Here we have provided a relevant `.gitignore` file in the skeleton. In general we want to exclude platform specific files, like the OSX .DS_Store files, any files that are automatically generated as well as files containing secrets such as API keys.

If you haven't done so already commit your changes to `index.js`:

1. Start by running 💻 `git status` in the terminal in the assignment directory to see how your modified files are reported.
2. Then add the modified files to stage them for the commit, i.e. 💻 `git add index.js`. The staging area now contains the files whose changes will be committed.
3. Run 💻 `git status` again to see the how staged files are reported.
4. Commit your changes with 💻 `git commit -m "Your pithy commit message"` (replace "Your pithy commit message" with a pithy but informative commit message, quotes are required). You can also skip the `-m` option. If you do so, `git` will open a text editor for you to write your commit message.
5. Run 💻 `git log` to see your commit reported.

Finally submit your assignment by pushing your changes to the GitHub classroom via 💻 `git push --all origin` and then submitting your repository to Gradescope as described here. You can submit (push to GitHub and submit to Gradescope) multiple times. The last submission before the deadline will be the one graded.

Grading

Assessment	**Requirements**
Needs revision	Some but not all tests as passing.
Meets Expectations	All tests pass, including linter analysis (without excessive deactivations).
Exemplary	All requirements for *Meets Expectations* and your implementation is clear, concise, readily understood, and maintainable.
