

Practical: Javascript

Initial Due Date: 2023-02-21 8:15AM

Final Due Date: 2023-03-31 4:15PM

Goals

- Get started with basic JavaScript and Node.js
- Get started with Git, GitHub classroom, and Gradescope

In this practical, you are going to create a simple JavaScript module to get a feel for some basic JavaScript principles and start getting familiar with our tools.

Prerequisite

Make sure you complete the steps on the [getting started](#) page. If you are using `nvm` and you have installed multiple versions of node, before using `node`, `npm` and associated tools, you need to make sure the correct version is activated if you have several version installed (■ `node -v` should return "", though the minor and patch numbers after the first dot may vary slightly).

Creating the module

Create a new package by first creating the package directory (call it `practical-01-username`, using your GitHub username) and then running ■ `npm init` *inside* the new directory, e.g.,

```
■ mkdir practical-01-mlinderm
■ cd practical-01-mlinderm
■ npm init
```

The `npm init` command will create the `package.json` file by asking you a series of questions. For most of the questions, you can accept the default. When it asks for the "test command", type `jest`. If your directory is a Git repository (not the case here) `npm` will automatically pull information from your Git repository to create the `package.json` file.

The `package.json` file can be edited like any other text file. You can open it directly in VSCode, but I find it more useful to open the entire directory as a project in VSCode, and then navigate to the file from there. There are two ways to do this. You can use the 'Open' option in the File menu and open the directory, or, if you have the command line tool installed, you can type ■ `code .` in the terminal to open the current directory (in the shell `.` is a shortcut for current directory and `..` is a shortcut for the parent directory).

Add two properties to the `package.json` file:

- `"private": true` This prevents you from [accidentally publishing](#) this package to npm
- `"type": "module"` This will make your code act as an ES module. The details of this aren't important, but it will mean that you can use the same style of code as we will later in the course.

After your manual editing your initial `package.json` file should look something like the following:

Make sure to save package.json before moving on. Installing will try to update package.json for you, and if you haven't saved, you will create edit conflicts.

```
{
  "name": "practical-01-mlinderm",
  "version": "1.0.0",
  "private": true,
  "type": "module",
  "description": "CS312 Javascript practical exercise",
  "main": "index.js",
  "scripts": {
    "test": "jest"
  },
  "author": "Michael Linderman <mlinderman@middlebury.edu>",
  "license": "Apache-2.0"
}
```

We should get comfortable looking at [JSON files](#). JSON files are plain text file used for structured data (e.g., configurations, messages). It is effectively a JavaScript object written out, the name stands for "JavaScript Object Notation". However, there are some subtle differences to be mindful of:

- The only accepted **values** (right side of the :) are numbers, Booleans, strings, Arrays, or objects
- The **keys** (left side of the :) in a JSON file are always quoted strings (as opposed to JavaScript where you should not use quotes)
- There are no trailing commas allowed (in JavaScript we can have a trailing comma after the last element of an array or object)

If you even have problems with your `package.json` file being parsed, check for the second and third of these.

Installing dependencies

For most JavaScript projects, you will install a collection of dependant packages. The tool we used above (`npm`) is the Node Package Manager (it isn't the only option, but it is the one we will be using). You won't need any packages for this assignment, but I will to evaluate it. So, I am going to have you install it so you get a feel for the process. You are going to install a tool called `jest`, which helps run automated tests (we will learn about how to use it shortly).

On the command line, execute `npm install -D jest`.

This will take a few moments, and then you should see a message that looks like (although the details will differ and your version may be newer).

```
added 278 packages, and audited 279 packages in 14s
```

```
30 packages are looking for funding
  run `npm fund` for details
```

```
found 0 vulnerabilities
```

Now look inside of `package.json` again. You should see that it now has a new section:

```
"devDependencies": {  
  "jest": "^29.4.3"  
}
```

You should also see a new file `package-lock.json` and a new sub directory named `node_modules` into which `jest` and all of its dependencies have been installed.

We used the `-D` flag to install the package as a "development dependency". These are packages that we need when we are building our modules, but should not be included in the deployment.

Writing some functions

Let's write our first Javascript functions. Create a new file named `index.js` in the directory (or just open it if it was automatically created for you).

Summation

Write a function that sums all of the values in an array (you can assume the values are numerical). For our version, let's write the code using a conventional `for` loop. Create a new function called `summationFor`.

```
export const summationFor = (list) => {  
  // ...  
};
```

[Copy](#)

For loops in JavaScript look a lot like they do in Java/C/etc... To iterate over the indices of a list, we would use:

```
for (let i = 0; i < list.length; i++) {  
  // ...  
}
```

[Copy](#)

Using the snippet above, complete the implementation of your function.

The `export` keyword makes your function visible outside of the file, which we will need for testing.

Testing the function

For our first tests we will run the Node.js interpreter (and it use it like the Python or other interpreters). Invoke `node` on the command line.

Tip: VSCode has an integrated command line that you can open at the bottom of the window. You will find it listed as 'Terminal' in the 'View' menu.

To load your code into `node`, type `let p1 = await import("./index.js")` (we will learn more about `await` later and why we need to use this particular import style in this context).

This creates a new object called `p1` (you can call this anything) and attaches all of your exports to it as properties. You can now run the functions by invoking them on `p1` (e.g., `p1.summationFor([1,2,3])`). Test it out with a couple of Arrays and make sure it works.

If you make a change to `index.js`, you will need to re-import it into `node`. One simple approach is just to exit (see below) and restart node. Alternately, to re-import the file we will unfortunately need to work around some internal import caching performed by NodeJS. Since this cache is based on the filename, we can append a changing query parameter to the import file name (which is really a URL) so that it is treated as a different file (with respect the cache), but loads the same file from your computer. For example:

```
p1 = await import(`./index.js?v=${Date.now()}`)
```

To leave `node`, you can invoke `.exit` or use the keyboard shortcut `^d` (Ctrl-d).

Summation, take 2

As you know from class we wouldn't really use a `for` loop for this. Write a second function called `summationForEach`. You will replace the `for` loop with the `forEach` function. The `forEach` function is a [method of arrays](#). Recall that it is a *higher-order* function that takes in a function as an argument. The `forEach` function calls the function you pass in once per element of the array, passing the element in as the first argument of the provided function.

Test your function in `node` to make sure it works.

Summation, take 3

Even this approach is more iterative than we really would use for a problem like this. I would like you to write this function one more time, but this time instead of `forEach`, you are going to use `reduce`. `reduce` is another high-order method of arrays. Its job is to run a function on every element of an array, accumulating a result – in essence, the array is *reduced* down to a single value. The function you pass into `reduce` should take two arguments: 1) the accumulated value and 2) the current element in the array.

Write a new function called `summationReduce`, which uses the `reduce` method to sum the list. Note that `reduce` is going to do all of the work, so, `summationReduce` could actually use the implicit `return` version of the fat arrow syntax (as could the reducer function that you pass to `reduce`). Test your new function.

When you are happy that the new function is working, try running it on an empty array. You will probably get something that looks like this:

```
> p1.summationReduce([])
Uncaught TypeError: Reduce of empty array with no initial value
    at Array.reduce (<anonymous>)
    at p1.summationReduce (REPL1:1:40)
```

The `reduce` method works by starting with the first element in the array as its initial value. If there isn't one... an error is thrown. We can fix this by adding a second argument to the `reduce` function. This becomes the starting value for the reduction. Add in a 0 as the initial value (add it to `reduce`, **not** to the reducer function, which already has two arguments).

Try it again, verifying that your function works correctly on empty arrays.

Mapping

While we are thinking about higher-order functions, let's try a `map` example.

Write a `decorate` function that returns an array of "decorated" strings.

```
> p1.decorate(['Sam', 'Penny', 'Jordan'])  
[ '-<< Sam >>-', '-<< Penny >>-', '-<< Jordan >>-' ]
```

Any time that you have an array of values and you need a second array of values reflecting some transformation of each value, you should think `map`. The helper function you pass to `map` takes in one value and returns the transformed value.

A common error is to think about the array as a whole. With `map` you only need to think about what should happen to a single value and write a function to do that. So, start by writing a function that takes in a string and returns a "decorated" string. Then, just pass this new function to `map` and it will do the rest, applying it to each value in the array and loading the results into a new array for you.

A helpful tool here is that JavaScript [template literal](#). This allows you to insert JavaScript expressions into a string. To make a template literal, use back ticks (`) instead of single or double quotes. You can then include expressions in the string by surrounding them with `${}`.

```
const x = 5;  
const s = `The value of x is ${x}`;  
  
console.log(s); // this will print out 'The value of x is 5'
```

Write and test the `decorate` function. Don't change the decoration as it will break the automated tests.

Getting Started with Git

Now that you have implemented your module, we want to turn the module into a Git repository.

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 *main* branch untouched until your new feature is ready to be *merged* back into the main.

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.

We will use Git and GitHub (in concert with Gradescope) to submit your work. Keep in mind the “distributed” in distributed VCS. Until you have pushed your changes to GitHub (and submitted your repository to Gradescope) your work is *not* turned in.

We will create a new Git repository with the command line. Make sure that the current working directory of your shell is the project folder, and then type  `git init`. This will create a new Git repository in your current directory (stored in a hidden directory called `.git`.)

Creating a commit (with new files or changes to existing files) is a two step process. First you **stage** the changes you want to preserve ( `git add`) and then you **commit** the changes, which saves them in the repository ( `git commit`).

Before we do this with your files, however, I should note that sometimes there are files that we **do not** want in the repository. These tend to be files that we can recreate later, or, in our case, other people's code that we can always download again (i.e., the contents of `node_modules`). We **really** don't want `node_modules` to sneak into the repository. The directory can get quite large, and it fouls up Gradescope to no end if you include it.

As you will see below, if we are careful, we can avoid including `node_modules`, but we want a solution that we can set once and forget. We can configure Git with a file called `.gitignore`. Git consults this file first before looking for changes in the project.

Create a new file called `.gitignore` and add the following to it:

```
# See https://help.github.com/ignore-files/ for more about ignoring files.
```


Copy

```
# Dependencies
/node_modules
```

```
# Misc
/.vscode
.DS_Store
```

This will ignore `node_modules` as well as some other invisible files that VSCode and MacOS use to store data about the current directory.

Once you've saved that file, we can walk through the process of adding files to the repository..

1. Make sure your files are all saved
2. Stage your files with  `git add .`. This adds all of the files in the current directory. Be careful of this! I usually recommend that you target the files you actually want to stage, but in this case, we already told Git

what to ignore and we want everything else. *Your files are not yet saved into the repository, they are only staged!*

3. Check that the files are staged with `git status`. This isn't a required step, but I recommend using it often to see the state of your files. It should show that all of your files are new and staged.
4. Commit your changes with `git commit -m "My first commit"`. This loads your files into the repository. The `-m` allows you to add a message associated with your commit. This should be short but informative so you can tell what you were trying to accomplish with the commit in case you need to go back to it later.
5. Check the status again – there should be no changes.

Let's do that one more time

1. Create a comment at the top of the file and add your name. (JavaScript comments are the same as in Java and C).
2. Type `git status` to see that there is a modified (but not staged) file.
3. Type `git add index.js`.
4. Type `git status` again to see how the status has changed.
5. Now commit the changes to the repository with `git commit -m "Added name to the top"`.

We can also look at the history of the repository with `git log`. You should see both of your commits in the log output.

You should work on getting comfortable with the command line, but you will find that VSCode actually has a reasonable git client built in. You will find it by clicking the 'Source Control' icon (it looks like a graph) on the left).


Multiple version of your code are now stored in your local repository. To turn your code in, you need to get a duplicate of your repository onto GitHub. The conventional way to do that would be to go to the GitHub site and create a new repository there. However, for this class we are using something called GitHub Classroom, which will automate this process of creating repositories for everyone in class

Click through to the GitHub [classroom assignment](#). Clicking through to the assignment will create a private repository for you with the name "practical-01-<Your GitHub username>.git", e.g. "mlinderm.git". You can view your repository at <https://github.com/csci312a-s23/practical-01-<Your GitHub username>> (click the link provided by GitHub classroom).

On the front page of your new repository, it may list some options about how to get started. If those instructions appear, follow the directions for existing repositories. If not, follow the steps below.

1. `git remote add origin <repository URL>`, where `**` is the URL for your repository at GitHub (don't be the one that actually types out `""`). You can find this URL by clicking on the green "Code" button and copying one of the URLs. There are two kinds of URLs: HTTPS and SSH. If you set up SSH for your GitHub account use the second, if not use the first (HTTPS). This will enable you to push to the remote repository using "origin" as a short-hand name.
2. `git branch -M main`. This is *not* actually an essential step for connecting the two repositories. This renames the primary branch of the repository from the git default of "master" to the more inclusive "main".
3. `git pull --rebase origin main`. This retrieves starter code created by GitHub classroom that you don't have (recall the distributed nature of Git). This should not be needed in the future when you start by

cloning the repository GitHub classroom creates for you. Note that to use the HTTPS interface you will need to create a [GitHub access token](#) before this step that you will use instead of your password when prompted.

4.  `git push -u origin main`. This tells git to copy changes from your current repository to "origin", which is the repository on GitHub.

If you reload the GitHub page, you will see it now lists your code.

Now, you can submit the practical to Gradescope as described [here](#)

Successfully submitting your assignment is a multi-step process: 1) Commit and push your changes to GitHub, 2) Submit your repository to the Gradescope assignment. Make sure you complete both steps (in order).

Grading

Required functionality:

- Create Node.js project
- Install `jest`
- Write three versions of summation using a loop, `forEach` and `reduce`
- Write `decorate` using `map`
- Setup git and GitHub
- Submit to Gradescope

Recall that the Practical exercises are evaluated as "Satisfactory/Not yet satisfactory". Your submission will need to implement all of the required functionality (i.e., pass all the tests) to be Satisfactory (2 points).

© Michael Linderman and Christopher Andrews 2019-2023. Last modified at: 2023-02-16 20:41:21 -0500.