Week 3 Study Guide

Asynchronous JS Lesson Learning Objectives

1. Identify JavaScript as a language that utilizes an event loop model

2. Identify JavaScript as a single threaded language

   JavaScript is a single threaded language, which can only run one statement of code at at a time.

   The JavaScript runtime can only do one thing at a time. In order to write performant, responsive code we utilize callbacks in order to perform slow tasks asynchronously. This keeps the single thread free to run code that's ready to run now.

   The event loop facilitates the management of callbacks and asyncronous behaviour within the single threaded runtime.

   Bonus video [25 min]: What the heck is the event loop anyway?

3. Describe the difference between asynchronous and synchronous code

   Synchronous code (often called iterative code) is code that runs in linear order, one expression following the next, from beginning to end.

   Asynchronous code is code that we schedule to run later. Usually, we have to wait for some other code to complete before we can proceed with our current code, so we wrap it in a callback, and ask JavaScript to return asynchronously to our callback.

4. Execute the asynchronous function setTimeout with a callback.

```
setTimeout(function() {
    console.log('TIME!');
}, 1000);
```

5. Given the function

```
function asyncy(cb) {
    setTimeout(cb, 1000);
    console.log("async")
}
```

   and the function

```
function callback() {
    console.log("callback");
}
```

predict the output of `asyncy(callback);`

```
> asyncy(callback);
'async'
'callback'
```

6. Use setInterval to have a function execute 10 times with a 1 second period. After the 10th cycle, clear the interval.

```
function tenIntervals( ) {
    let count = 0;
    let handler = setInterval(function() {
        count++;
        console.log('execute');
        if (count == 10) {
            clearInterval(handler);
        }
    }, 1000);

}
tenIntervals();
```

7. Write a program that accepts user input using Node's readline module

```
const readline = require('readline');

const rl = readline.createInterface({
    input: process.stdin,
    output: process.stdout
});

console.log("hello");

rl.question('What would you like to say? ', (answer) => {
    console.log('You says: ' + answer);
    rl.close();
});
```

Node.js Lesson Learning Objectives

1. Define NodeJS as distinct from browser based JavaScript runtimes.

   JavaScript (like most programming languages) has mutltiple runtimes, each of which is a full implementation of the language and supporting libraries. With JavaScript, the two implementations (runtimes) that we care about are V8 (used in Chrome, and also the Node.js runtime) and Spidermonkey (used in Firefox).

2. Write a program that reads in a dictionary file using node's FS API and reads a line of text from the terminal input. The program should 'spell check' by putting asterisks around every word that is NOT found in the dictionary.

```
const readline = require('readline');
const fs = require('fs');
let dictionary = [];

const rl = readline.createInterface({
   input: process.stdin,
   output: process.stdout
});

function askQuestion() {
    rl.question('What would you like to spell check? ', (answer) => {
        let wordsToCheck = answer.split(' ');
        let result = [];
        for (let i =0; i < wordsToCheck.length; i++) {
            if (dictionary.includes(wordsToCheck[i])) {
                result.push(wordsToCheck[i]);
            } else {
                result.push("*" + wordsToCheck[i] + "*");
            }
        }
        rl.close();
        console.log(result.join(' '));
    });
}


fs.readFile('dictionary.txt', 'utf8', (err, data) => {
   if (err) {
       console.log(err);
       return;
   }
   console.log(data);
   dictionary = data.split('\n');
   //using callback chaining
```

```
    askQuestion();
});
```

Git Lesson Learning Objectives

1. Use Git to initialize a repo

   ```
   git init
   ```

   This will create a `.git` hidden folder inside your new repository.

2. Explain the difference between Git and GitHub

   Git is a distributed version control system that allows us to build up patch sets and changes.

   GitHub is a company that provides *git* repository hosting as well as project management features like code review, issues, wiki, etc.

3. Given 'adding to staging', 'committing', and 'pushing to remote', match attributes that apply to each.

   Adding to Staging:

   - Changes are added to the current patchset
   - Changes are not yet committed
   - Changes can be reviewed with `git diff --cached`

   Committing:

   - Create a commit for a set of related code changes (a patchset)
   - A git history is made up of a history of many commits
   - Use `git commit` to create a commit from all the changes that are currently in staging

   Pushing to Remote:

   - Pushing the changes from your local github repository to a remote repository
   - Until changes are pushed, no other team members can see them
   - Code must be pushed to a remote in order to create a pull request

4. Use Git to clone an existing repo from GitHub

   After discovering the repository url (from GitHub, Bitbucket, etc. or if someone sends you a link to their privately hosted git repository)

   ```
   git clone <repo_url>
   ```

5. Use Git to push a local commit to a remote branch

   The changes tracked by your local branch called `branch_name` can be pushed to any remote with the `git push` command.

   ```
   git push <remote> <branch_name>
   ```

6. Use git to make a branch, push it to github, and make a pull request on GitHub to merge it to master

   Assuming you are working on an existing project. First create a branch

   ```
   git checkout -b <my_branch>
   ```

   Make your changes, add them and commit them

   ```
   git add -p
   git commit -m 'my awesome changes.'
   ```

   Push your newly created branch.

   ```
   git push origin <my_branch>
   ```

   Then point your browser at the GitHub repository, and follow the onscreen prompts to create a pull request.

7. Given a git merge conflict, resolve it

   Once you've done the mental work of correcting the conflicting errors, use `git add` and `git commit` to commit the resolved code.

8. Match the three types of git reset with appropriate descriptions of the operation.

   Hard (`git reset --hard <ref>`):

   - resets destructively, changing your code back to the commit identified by ref
   - deletes any code that conflicts with the identified commit

   Soft (`git reset --soft <ref>`):

   - resets non-destructively. Resets your head back to ref.
   - Does not change your code changes or 'staged' code at all

Mixed (`git reset <ref>`):

- Resets mostly-non-destructively. Resets your head back to ref.
- Staged changes are moved out of the staging area

9. Use Git reset to rollback local-only commits.

```
git reset --hard HEAD
```

10. Identify what the git rebase command does

`git rebase` allows us to rewrite the history of the current branch my changing commit contents, or adding commits to our history and replaying all the commits since that moment on top of them. Rebase allows us to clean up mistakes in our history, or to avoid adding "merge commits" by adding the commits from another branch into our branches' history.

11. Use git diff to compare a local 'staging' branch and 'master' branch.

```
git checkout staging
git diff master
```

git diff by itself shows you the difference between your current branch and any unstaged changes you have in your working directory.

```
git diff
```

12. Use git checkout to check out a specific commit by commit id

This will leave your git repository in a "headless" state, which you cannot apply commits on top of.

```
git checkout <ref>
```

Command Line Interface Basics Lesson Learning Objectives

1. Given a folder structure diagram, a list of 'cd (path)' commands and target files, match the paths to the target files.

Given a sample filesystem:

```
.
├── home
│   └── idbently
│       └── development
│           └── appacademy
├── rmdir
├── usr
│   ├── bin
│   └── local
│       └── bin
└── var
    ├── lib
    └── log
```

And knowing:

```
$ pwd
/home/idbentley
```

I can navigate to the `appacademy` directory with the command:

```
cd development/appacademy
```

2. Create, rename, and move folders using unix command line tools.

```
$ pwd
/home/idbentley/development
$ ls -a
. ..
```

Knowing that we are in an empty development directory, I want to create a new directory for my project `ChewbaccaCat`. I know I can use the `mkdir` tool, so I try

```
$ mkdir ChewbaccoCat
$ ls -a
. .. ChewbaccoCat
```

Dern, typos! I can use `mv` (i.e move) to rename a file or folder, however.

```
$ mv ChewbaccoCat ChewbaccaCat
$ cd ChewbaccaCat
$ pwd
/home/idbentley/development/ChewbaccaCat
$ git init
```

3. Use grep and | to count matches of a pattern in a sample text file and save result to another file.

   One option using grep with the `-c` (count) opt

   ```
   $ grep -c 'pattern' infile.txt > countResults.txt
   ```

   Or we can pipe `|` the output of grep into the word count command `wc -l` to count the number of matched lines

   ```
   $ grep "pattern" infile | wc -l > countResults.txt
   ```

4. Find what -c, -r, and -b flags do in grep by reading the manual.

   - `-c` stands for "count". It will limit the output of grep to just the number of matches, not the matches themselves.
   - `-r` is unsurprisingly "recursive", and it will tell grep that it should search the contents of any directories it receives as input.
   - `-b` stands for "byte-offset" and it will tell you the line number, and position of grep's matches.

5. Identify the difference in two different files using diff.

   ```
   $ diff file1 file2
   ```

   Easy as pie

   Optionally you can get 'unified' diff output by using the `-u` command line option (similar to `git diff` output)

   ```
   $ diff -u file1 file2
   ```

6. Open and close nano with and without saving a file.

   To open nano, we point the `nano` command at the file we wish to open

```
$ nano countResults.txt
```

Nano uses `Ctrl + <command>` keyboard shortcuts to utilize it's functions.

We use `Ctrl + o` to save a file - Nano will prompt you for the buffer name (Nano calls files "buffers" for historical reasons).

To exit Nano without saving, you can use `Ctrl + x`.

7. Use 'curl' to download a file.

   Without specifying `-o` curl downloads the file to STDOUT (standard output i.e. your terminal)

   ```
   $ curl https://www.google.com/robots.txt
   ```

   If we want to save the file, we can use the the `-o` option and give it a filename

   ```
   $ curl https://www.google.com/robots.txt -o robit.txt
   ```

8. Read the variables of $PATH.

   To read the current value of path, use the `echo` command

   ```
   $ echo $PATH
   ```

   The `$PATH` variable is split on `:` characters, resulting in an array of paths, which should be read from left to right. When your OS is looking for a command to run, it will walk through this array, checking each path in turn.

   You can use the `which` command to search the path for a command have it print out the full path to it. For instance to locate where `grep` exits:

   ```
   $ which grep
   /usr/bin/grep
   ```

9. Explain the difference between .bash_profile and .bashrc.

   Note: This LO will not be on the assessment.

However, `.bash_profile` is executed when bash is run as a "Login" shell with the `-l` command line option, while `.bashrc` is executed when bash is ran without `-l`

10. Create a new alias by editing the .bash_profile.

    For example:

    ```
    alias la='ls -a'
    alias gl='git log --oneline'
    ```

11. Given a list of common scenarios, identify when it is appropriate and safe to use sudo, and when it is a dangerous mistake.

    sudo runs a command as the `root` or super user of the system.

    ```
    $ sudo rm <path>
    ```

    Any use of sudo rm is potentially a dangerous mistake! However commands like `rm` are especially dangerous because they remove files!

    ```
    $ sudo ls /somefolder
    ```

    Running a non-destructive command like `ls` would be safer to run with sudo.

    ```
    $ sudo apt install <package>
    ```

    Installing system packages is a safe and necessary use of sudo

    ```
    $ sudo npm install -g mocha
    ```

    Installing npm packages with sudo is almost always gonna mess up your node installation, because in this class we use `nvm` to manage our node.js installation and `nvm` installs the files into your home directory (`~`) so sudo isn't needed.

12. Write a shell script that greets a user by their $USER name using echo.

    We write our simple script to `whoami.sh`

    The shebang(`#!`) line specifies which shell we want the script to run under.

Commonly we will use `#!/bin/sh`, but you might sometimes use `#!/bin/bash` or `#!/bin/zsh` if you are using specific features of those shells. `/bin/sh` can be thought of as a basic shell that is useful for most shell scripts. (it implements the POSIX shell scripting standards)

```
#!/bin/sh
USR=`whoami`
echo "Hello " $USR
```

13. Use chmod to make a shell script executable.

    Now we want to run `whoami.sh` so we add the executable (`x`) permission to the user(`u`)

```
$ ./whoami.sh
zsh: permission denied: ./whoami.sh
$ chmod u+x whoami.sh
$ ./whoami.sh
Hello idbentley
```

Recursion Lesson Learning Objectives

1. Given a recursive function, identify what is the base case and the recursive case.

```
// Computes the product of the numbers in an array
function product(arr) {
    if (arr.length === 1) {
        return arr[0];
    } else {
        return arr[0] * product(arr.slice(1));
    }
}
```

    In this case, the body of our if block, when `arr.length === 1` is our *base case*.

    The *recursive case* is where arr.length is NOT 1, and this means our *recursive step* is `arr[0] * product(arr.slice(1))`.

2. Explain when a recursive solution is appropriate to solving a problem over an iterative solution.

    There are no hard and fast rules as to when recursive solutions are appropriate compared to an iterative solution. Your comfort with each type of programming will inform which paradigm you prefer. List comprehension functions, like reduce, map and filter are implemented very cleanly with recusion, and indeed were introduced by functional languages before being adopted by a whole host of predominantly iterative languages including JavaScript.

Through problems, like our naive recursive solution to Fibonacci, we learn that there are definitely times when we a problem can be expressed cleanly as a recursive algorithm, but will have very poor performance. We clearly need to consider more than just lines of code and clarity of expression when deciding whether or not to use a recursive solution.

There are certain classes of problems: i.e. problems that traverse nested arrays, objects, graphs or trees (that is, data with recursive structure) which lend themselves especially well to recursive solutions.

As we continue to gain experience with algorithms in this course, we will learn about the complex topic of runtime complexity - the analysis of the performance of our code. At that time, we will have an understanding of the principals of code performance, which will enable us to discuss the merits of various algorithms, both iterative and recursive, in a more concrete fashion."

3. Write a recursive function that takes in a number, n, argument and calculates the n-th number of the Fibonacci sequence.

```
function fibonacci(n) {
    if (n <= 2) {
        return 1
    }
    return fibonacci(n - 1) + fibonacci(n-2);
}
```

4. Write a function that calculates a factorial recursively.

```
function factorial(num) {
    if (num === 1) {
        return 1;
    }
    return factorial(num-1) * num;
}
```

5. Write a function that calculates an exponent (positive and negative) recursively.

```
function exponent(num, power) {
    if (power < 0) {
        return (1 / exponent(num, Math.abs(power)));
    }

    if (power === 1) {
        return num
    }
```

```
    return num * exponent(num, power - 1);
}
```

6. Write a function that sums all elements of an array recursively.

```
function sum(arr) {
    if (arr.length == 1) {
        return arr[0];
    }
    return arr[0] + sum(arr.slice(1));
}
```

7. Write a function that flattens an arbitrarily nested array into one dimension.

```
function flatten(arr) {
    let res = [];
    arr.forEach(el => {
        if (Array.isArray(el)) {
            let flattened = flatten(el);
            res = res.concat(flattened);
        } else {
            res.push(el);
        }
    });
    return res;
}
```

8. Given a buggy recursive function that causes a RangeError: Maximum call stack and examples of correct behavior, debug the function.

   When we have a recursive solution that overflows the call stack, we know that the problem must be in our **base case** or our **recursive step**. Our algorithm will *not make progress* if our recursive step is missing, and our algorithm will not know when it *has completed* without a base case.

JS Trivia Lesson Learning Objectives

1. Given a code snippet of a unassigned variable, predict its value.

   Using `let`:

```
let james;
console.log(james); // undefined
```

Using `const`;

```
const goodbye;
console.log(goodbye); // SyntaxError: Missing initializer in const
declaration
```

Inside a function block using `let`

```
    let func1 = () => {
        let hello;
        console.log(hello);
    };
    func1() // undefined
```

Inside a function, accessing before declaration using `let`

```
    let func2 = () => {
        console.log(hello);
        let hello;
    };
    func2(); // ReferenceError: Cannot access 'hello' before
initialization
```

Inside a function block, accessing before declaration using `var`

```
    let func3 = () => {
        console.log(hello);
        var hello;
    };
    func3() // undefined
```

2. Explain why functions are “First Class Objects” in JavaScript

    - You can store the function in a variable
    - You can pass the function as a parameter to another function
    - You can return the function from a function
    - Functions are considered first class objects in JavaScript because in JavaScript they are treated like any other variable.

3. Define what IIFEs are and explain their use case

An Immediately-Invoked Function Expression(IIFE) is a function that is called immediately after it had been defined.

```
let result = (function () {
    let event = "Coding Party"
    return event;
})();

console.log(result); // prints "party!"
```

One of the main advantages gained by using an IIFE is the very fact that the function cannot be invoked after the initial invocation.

4. (Whiteboarding) Implement a closure

   We can return a function from a function. The inner function will close over any variables and parameters of the outer function.

   In this example the parameter `adder` is held in a closure by the inner function that returns `num + adder`

```
    function dynamicAdder(adder) {
        return function (num) {
            return num + adder;
        };
    }

    let addThree = dynamicAdder(3);
    console.log(addThree(6)); // 9
    console.log(addThree(9)); // 12
```

5. Identify JavaScript's falsey values

   The javascript falsey values are:

   - undefined
   - ""
   - 0
   - NaN
   - null
   - false
   - 0n

6. Interpolate a string using back-ticks

```
function welcome(name, time) {
    console.log(`Hi ${name}! Isn't it a wonderful ${time}.`);
}

welcome("Victoria", "evening")
```

7. Identify that object keys are strings or symbols

```
    const code = Symbol("javascript");
    const languages = {
        [code]: "JS",
        Python: "P"
    };

    console.log(Object.keys(languages));
    console.log(Object.getOwnPropertySymbols(languages));

    console.log(Symbol("C++") === Symbol("C++"));
```

8. A primitive type is data that is not an object and therefore cannot have methods(functions that belong to them).

   Primitive types are `immutable` unlike reference types which are `mutable`. Objects are reference types, all other types are primitive types.

   The exception to the object rule is Strings which have methods, but only because JavaScript temporarily converts them to objects when you try to use a string method on a string.

9. Given a code snippet where variable and function hoisting occurs, identify the return value of a function.

   using `let` it behaves as expected, and trying to access a variable before it is initialized fails and throws a "Can't access before initialization" error.

```
console.log(goodNight()); // ReferenceError: Cannot access 'goodNight'
before initialization

let goodNight = function goodNight() {
    return "Good Night!";
};
```

   using `var` throws a different error, because var is declared before we try to call `goodNight`

```
```

```
console.log(goodNight()); // TypeError: goodNight is not a function

var goodNight = function() {
    return "Good Night!";
}
```

using `let` before declaring a function results in an error because the identifier `hello` has already been declared by the `let` keyword.

```
let hello = "hello";

function hello(num) { // SyntaxError: Identifier 'hello' has already been
declared
    console.log("hello!");
}

console.log(hello);
```

functions can be declared after we call them because they are declared in a compile time step, or another way to say it, is the function declaration is hoisted.

```
console.log(shoutWord("apple")); // APPLE

function shoutWord(word) {
    return word.toUpperCase();
}
```

Using var with a function declaration with the same identifier results in the initialization of `hello` overwriting the function declaration, so we get an error because `hello` is not a function, it's a string. Yet another reason not to use `var`

```
var hello = "Hello";
function hello() {
    console.log("hello");
}

hello(); // TypeError: hello is not a function
```